

CodeWarrior™
Development Tools
Metrowerks Enterprise C
Compiler User's Manual

Revised: 2002/11/21

Metrowerks, the Metrowerks insignia, and CodeWarrior are registered trademarks of Metrowerks Corp. in the US and/or other countries. All other trade names, trademarks and registered trademarks are the property of their respective owners.

Intel and Pentium are registered trademarks of Intel Corporation.

Windows is a registered trademark of Microsoft Corporation.

© Copyright. 2002. Metrowerks Corp. ALL RIGHTS RESERVED.

Metrowerks reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Metrowerks does not assume any liability arising out of the application or use of any product described herein. Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft or automobile navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.

Documentation stored on electronic media may be printed for personal use only. Except for the forgoing, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

ALL SOFTWARE, DOCUMENTATION AND RELATED MATERIALS ARE SUBJECT TO THE METROWERKS END USER LICENSE AGREEMENT FOR SUCH PRODUCT.

How to Contact Metrowerks:

Corporate Headquarters	Metrowerks Corporation 9801 Metric Blvd. Austin, TX 78758 U.S.A.
World Wide Web	http://www.metrowerks.com
Ordering & Technical Support	Voice: (800) 377-5416 Fax: (512) 997-4901

Table of Contents

1 Introduction	11
Overview of the Metrowerks Enterprise C Compiler	11
The Cross-File Optimization Approach	12
Compiling Applications	12
The Compiler Shell Program	13
Stages in the C Compilation Process	13
2 Getting Started	17
Overview of Creating and Running a Program	17
Creating and Running a Program	17
3 Using the Metrowerks Enterprise C Compiler	19
The Shell Program	19
The C Compilation Process	20
Cross-File Optimization	22
File Types and Extensions	25
Environment Variables	27
Invoking the Shell	27
Shell Control Options	28
Option Summary	29
Controlling the Behavior of the Shell	33
Specifying Preprocessing Options	35
Overriding Input File Extensions	37
Output Filename and Location Options	38
Specifying C Language Options	39
Passing Options Through to Specific Tools	40
Setting the Options for Listings and Messages	41
Specifying the Hardware Model and Configuration	43
Language Features	45
C Language Dialects	45
Types and Sizes	57
Fractional and Integer Arithmetic	64
Intrinsic Functions	67
Pragmas	75
Predefined Macros	91

4	Interfacing C and Assembly Code	93
	Inlining a Single Assembly Instruction.	93
	Inlining a Sequence of Assembly Instructions.	94
	Guidelines for Inlining Assembly Code Sequences	94
	Defining an Inlined Sequence of Assembly Instructions	95
	Calling an Assembly Function in a Separate File	100
	Writing the Assembly Code	101
	Calling the Assembly Function	102
	Integrating the C and Assembly Files	103
	Including Offset Labels in the Output File	103
5	Optimization Techniques and Hints	107
	Optimizer Overview	107
	Code Transformations	107
	Basic Blocks	108
	Linear and Parallelized Code	108
	Optimization Levels and Options	110
	Using the Optimizer	112
	Invoking the Optimizer	112
	Optimizing for Space	113
	Using Cross-File Optimization	113
	Optimization Types and Functions	114
	Dependencies and Parallelization	114
	Target-Independent Optimizations	115
	Target-Specific Optimizations	129
	Space Optimizations	139
	Cross-File Optimizations	140
	Guidelines for Using the Optimizer	141
	Partial Summation Techniques	141
	Multisample Techniques	145
	General Hints	155
	Optimizer Assumptions	156
6	Runtime Environment	159
	Startup Code.	159
	Bare Board Startup Code	160
	C Environment Startup Code	161

Configuring Your Startup Code	162
Memory Models	163
Linker Command Files	164
Memory Layout and Configuration	165
Stack and Heap Configuration	166
Static Data Allocation	168
Configuring the Memory Map	168
Machine Configuration File	170
Application Configuration File	173
Calling Conventions	180
Stack Pointer	180
Stack-Based Calling Convention	180
Optimized Calling Sequences	182
Stack Frame Layout	183
Creating a Calling Convention	184
Interrupt Handlers	193
Frame Pointer and Argument Pointer	194
Hardware Loops	194
Operating Modes	194
7 Runtime Libraries	195
Character Typing and Conversion (ctype.h)	196
Testing Functions	196
Conversion Functions	197
Floating Point Characteristics (float.h)	197
Floating Point Library Interface (fltmath.h)	198
Integer Characteristics (limits.h)	201
Locales (locale.h)	202
Floating Point Math (math.h)	202
Trigonometric Functions	202
Hyperbolic Functions	203
Exponential and Logarithmic Functions	203
Power Functions	204
Other Functions	204
Nonlocal Jumps (setjmp.h)	204
Signal Handling (signal.h)	204
Variable Arguments (stdarg.h)	205

Table of Contents

Standard Definitions (stddef.h)	205
I/O Library (stdio.h)	206
Input Functions	206
Stream Functions	206
Output Functions	207
Miscellaneous I/O Functions	208
General Utilities (stdlib.h)	209
Memory Allocation Functions	209
Integer Arithmetic Functions	209
String Conversion Functions	210
Searching and Sorting Functions	210
Pseudo Random Number Generation Functions	211
Environment Functions	211
Multibyte Character Functions	211
String Functions (string.h)	212
Copying Functions	212
Concatenation Functions	212
Comparison Functions	213
Search Functions	213
Other Functions	214
Time Functions (time.h)	214
Time Constant	215
Process Time	215
Built-in Intrinsic Functions (prototype.h).	216

A Migrating from Other Environments 227

Code Migration Overview	227
Migrating Code Developed for DSP56600	228
Integer Data Types	228
Fractional Data Types	229
Floating Point Data Types	229
Pointers	229
Fractional Arithmetic	230
Inlined Assembly and C Code	231
Intrinsic Functions	231
Pragmas	232
Interrupt Handlers	232

Storage Specifiers	232
Miscellaneous	233
Migrating Code Developed for TI6xx	233
Data Types	233
Keywords	233
Pragmas	234
Inlined Assembly Code	234
Intrinsic Functions	234
B Modulo Addressing Example	235
C Induction-Related Loop Optimizations	243
Loop Detection and Normalization	243
Detection of hardware-mappable loops	243
Normalization of hardware loops	247
Loop-Invariant Code Motion	249
Scalarization.	250
Need and scope	250
Overview and goal	250
Assembly view and result	251
Pointer Promotion	253
Need and scope	253
Overview and goal	254
Assembly view and result	254
Single-loop Induction Process	255
Introduction	256
Simple induction variables	257
Multi-Step IV	260
Composition of IV	264
Wrap around variables	267
Monotonic variables	269
Modulo-induction	270
Simplification of redundant IV	279
Sequential Accesses and Related Optimizations.	283
Introduction	283
Basic transformation of sequential accesses, control strategy .	284
Simplification of redundant memory accesses	292

Table of Contents

Access packing (vectorization)	296
Cross-loop Induction	303
Introduction	303
A bestiary	308
D Loop Restructuring and Reordering	321
Definitions and Scope	321
Some definitions	321
Features of CodeWarrior for StarCore	322
Loop-Collapse	322
Overview and goal	322
Assembly view and result	324
Loop Peeling	325
Overview and goal	326
Assembly view and result	327
Loop Unrolling	327
Overview and goal	327
Assembly view and result	329
Partial Summation	332
Overview and goal	332
Assembly view and result	334
E Loop Restrictions	341
Limitations Concerning Single-Loop Induction	341
IV redefinition	341
Ambiguous definition due to function call	343
Multiple conditional induction	344
Second order induction	345
Limitations of Cross-Loop Mechanisms	345
General restriction on loop steps	346
Reused variables	347
Implicit cross-loop combination	349
Conditional inner loop	352
Bypassed inner loop	353
Limitations of Sequential Accesses and Packing.	357
Aliasing and conflicting interleaved read/write accesses	357
Aliasing and interprocedural effect	359

Missing initial alignment	360
Case Study: G729 cor_h Function.	361
Purpose and content of this chapter	362
Structure of function loop nests	362
Restrictions and solutions	363
Result	369
Index	371

Table of Contents

Introduction

This manual describes the features of the Metrowerks™ Enterprise C compiler.

NOTE This manual describes the features of the Metrowerks Enterprise C compiler, which is part of the CodeWarrior™ for the StarCore™ DSP product, and its command-line usage. For information on using the compiler with the CodeWarrior IDE, see the Targeting the StarCore™ DSP manual.

This chapter contains the following topics:

- [Overview of the Metrowerks Enterprise C Compiler](#)
- [The Cross-File Optimization Approach](#)
- [Compiling Applications](#)

Overview of the Metrowerks Enterprise C Compiler

A key feature of the Metrowerks Enterprise C compiler is its ability to generate code that is exceptionally compact, approaching the code density of the best RISC microprocessors while demonstrating high performance that is comparable to assembly code running on other DSPs. To achieve such a high performance, the compiler optimizes code for maximum parallelism in order to take full advantage of the core's multiple execution units.

In addition to its extensive optimization capabilities, the compiler offers a host of other features that make it ideal for DSP software development, including:

- Conformance to the ANSI C standard
- Intrinsic function support for ITU/ETSI primitives: saturating, non-saturating, and double-precision arithmetic

- Runtime libraries and environments
- Easy integration of assembly code into C code

The Cross-File Optimization Approach

The SC100 optimizer converts preprocessed source files into assembly output code, applying a range of code transformations which can significantly improve the efficiency of the executable program. The goal of the optimizer is to improve its performance in terms of execution time and /or code size by producing output code which is functionally equivalent to the original source code.

The method used by traditional compilers is to optimize each source file individually, before compiling the optimized code and submitting all the compiled files to the linker. Because all the necessary information is not available when files are optimized individually, the compiler must make various assumptions, and is unable to produce the most efficient result.

To ensure optimal performance, the optimizer can take advantage of visibility of as much of the application as possible. The SC100 global binder links all modules into a single module on which all optimizations can be performed. As a result of this approach, the performance of the optimizer is substantially improved, and the generated code is typically more efficient than if produced without cross-file optimization.

Compiling Applications

The SC100 compilation process consists of a series of steps, starting from the submission of source files and options to the C Front End (CFE), through the creation of Intermediate Representation (IR) files, the optimization of these files, and the output of optimized assembly code for linking into the final executable program.

You can perform all these processes in one single step, using the compiler shell program.

The Compiler Shell Program

The shell provides a one-step command-line interface, in which you specify the files to be processed for each compilation. At each stage, a different tool accepts the input files according to their file extensions, processes them, and outputs the transformed code for processing by the next development tool.

By default, the input files are progressed automatically through all the processing phases. The shell command line lets you select the exact development tools and processing stages that you require, and enables you to define any specific processing options, settings and/or default overrides that you need.

The options that you specify in the command line control the operation of the shell and of the tools used in the application development process. These options either affect the behavior of the shell itself or are dispatched to the different programs which the shell invokes.

The shell accepts a wide range of option types, including for example, those which perform specific actions, such as generating a list of included files, those which dictate how a source file should be treated, and those that control specific aspects of the C language features.

When you invoke the shell, the application development process is implemented automatically through all its various stages to the final production of the executable program.

Stages in the C Compilation Process

The following is an outline of the steps involved in compiling C source files into an executable program:

1. The shell is invoked with the list of the C source files and assembly files to be processed, and the various options to be applied.
2. The C Front End (CFE) identifies each C source file by its file extension, preprocesses the source files, converts the files into Intermediate Representation (IR) files, and passes these to the optimizer.
3. The high-level phase of the optimizer translates each intermediate representation file into an assembly ASCII file, and performs a number of target-independent optimizations.

Introduction

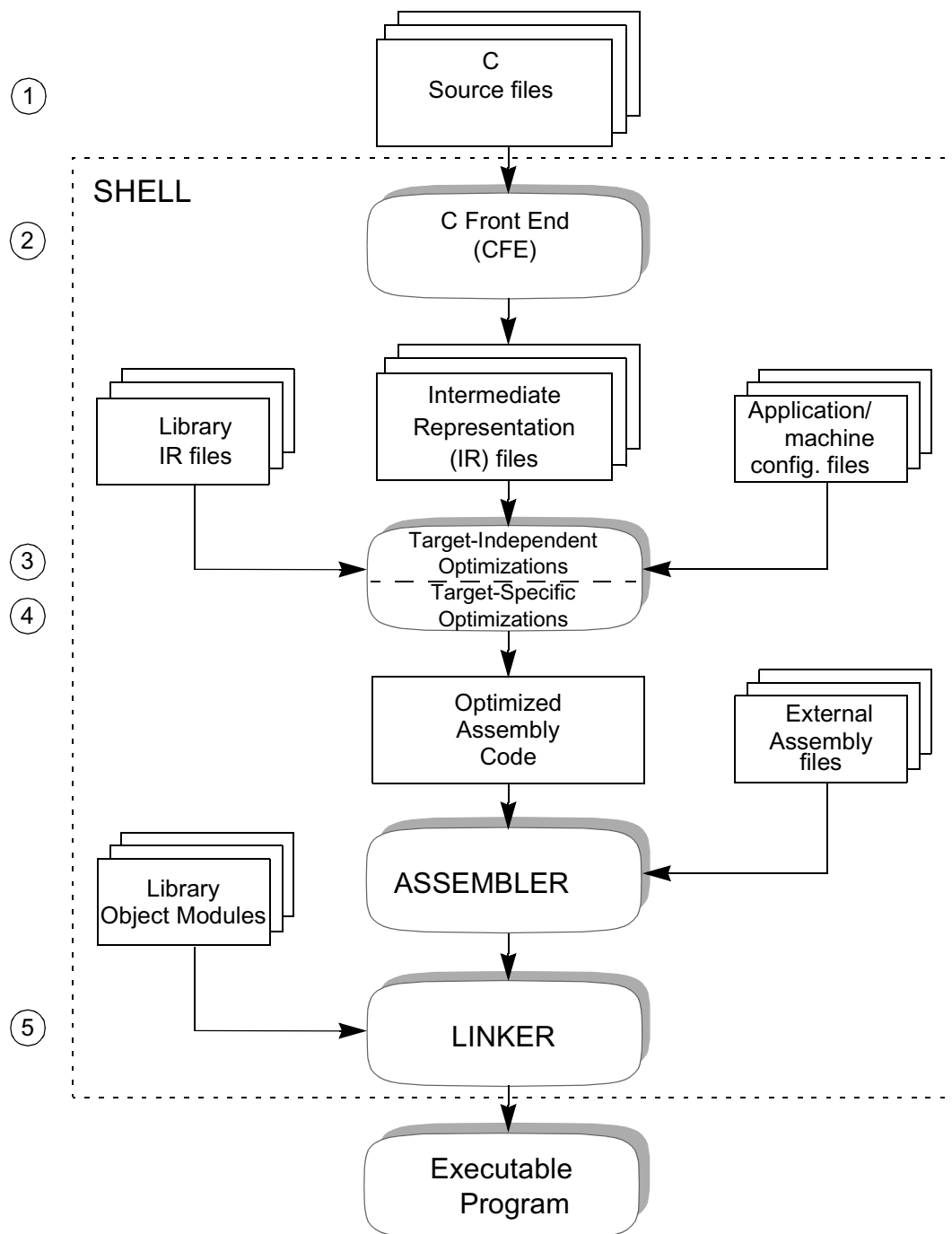
Stages in the C Compilation Process

Library files which have been created in IR form can be extracted by the optimizer, and included at this stage of processing. The optimization process also includes any relevant information contained in the application and machine configuration files.

4. The low-level phase of the optimizer carries out target-specific optimizations, and transforms the linear assembly code output by the previous phase into parallel assembly code.
5. At the end of the optimization, the optimized assembly files are output to the assembler, assembled together with any specified external assembly files, and from there output to the linker. The linker combines the assembly object files, extracts any required object modules from the library, and produces the executable application.

These stages are illustrated in the flow diagram shown in [Figure 1.1](#).

Figure 1.1 The SC100 C Compilation Process



Introduction

Stages in the C Compilation Process

Getting Started

This chapter explains how to build and run a simple program using the Metrowerks Enterprise C compiler.

This chapter contains the following topics:

- [Overview of Creating and Running a Program](#)
- [Creating and Running a Program](#)

Overview of Creating and Running a Program

The following general process describes how to create and execute a program from the command line:

1. Write the C source code, using the utility of your choice. In this example we will use a sample C source code file provided with your installation.
2. Compile and link the file, using the compiler shell.
3. Run the executable application that you have created.

Creating and Running a Program

Use the following steps to create and run a program from the command line:

1. Locate the file `hello.c` in the `$SCTOOLS_HOME/src/appnotes` directory, where `$SCTOOLS_HOME` is your installation directory. Copy the `hello.c` file into your working directory.

[Listing 2.1](#) shows the C source code contained in the `hello.c` file:

Listing 2.1 Sample source file: `hello.c`

```
#include <stdio.h>
```

```
void main()
```

Getting Started

Creating and Running a Program

```
{  
printf("Hello there!\n");  
}
```

- 2 Enter the following command to instruct the shell program to compile and link the program:

```
scc -o hello.eld hello.c
```

- 3 Run the executable program, by entering the following:

```
runsc100 hello.eld
```

The message `Hello there!` is displayed.

You successfully compiled, linked, and executed a program using the Metrowerks Enterprise C compiler.

Using the Metrowerks Enterprise C Compiler

This chapter explains how to use the Metrowerks Enterprise C compiler and describes the options and features that the compiler supports.

This chapter contains the following topics:

- [The Shell Program](#)
- [Invoking the Shell](#)
- [Shell Control Options](#)
- [Language Features](#)

The Shell Program

The shell program controls the processing of C source files and other files into an executable application, through the preprocessing, compilation, optimization, assembly and linking stages.

The shell provides a one-step command line interface, in which you specify the files to be processed for each compilation. At each stage a different tool accepts the input files according to their file extensions, processes them, and outputs the transformed code for processing by the next development tool.

By default, the input files are progressed automatically through all the processing phases. The command line lets you select the exact development tools and processing stages that you require, and define any specific processing options, settings and/or default overrides that you need.

- [The C Compilation Process](#)
- [Cross-File Optimization](#)

- [File Types and Extensions](#)
- [Environment Variables](#)

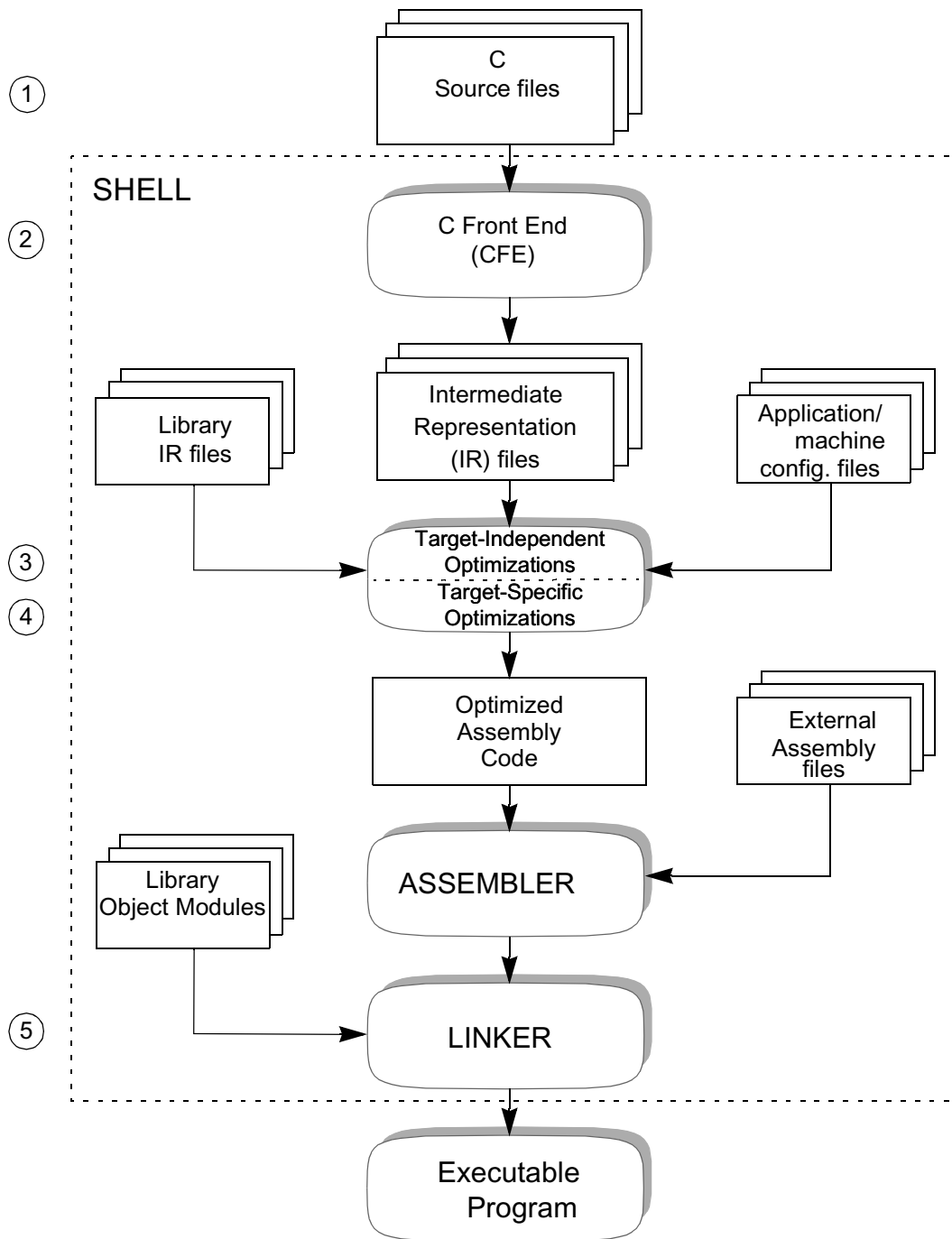
The C Compilation Process

The following is an outline of the process of compiling C source files into an executable program:

1. The shell is invoked with the list of the C source files and assembly files to be processed, and the various options to be applied.
2. The C Front End (CFE) identifies each C source file by its file extension, preprocesses the source files, converts the files into Intermediate Representation (IR) files, and passes these to the optimizer.
3. The high-level phase of the optimizer translates each intermediate representation file into an assembly ASCII file, and performs a number of target-independent optimizations. Library files which have been created in IR form can be extracted by the optimizer, and included at this stage of processing. The optimization process also includes any relevant information contained in the application and machine configuration files.
4. The low-level phase carries out target-specific optimizations, and transforms the linear assembly code output by the previous phase into parallel assembly code.
5. At the end of the optimization, the optimized assembly files are output to the assembler, assembled together with any specified external assembly files, and from there output to the linker. The linker combines the assembly object files, together with any specified external assembly files, extracts any required object modules from the library, and produces the executable application.

[Figure 3.1](#) shows the preceding process.

Figure 3.1 The C Compilation Process



Cross-File Optimization

The SC100 optimizer converts preprocessed source files into assembly output code, applying a range of code transformations which can significantly improve the efficiency of the executable program. The goal of the optimizer is to improve its performance in terms of execution time and /or code size by producing output code which is functionally equivalent to the original source code.

The method used by traditional compilers is to optimize each source file individually, before compiling the optimized code, and then submitting all the compiled files to the linker. Because not all the necessary information is available when files are optimized individually, the compiler must make various assumptions, and is unable to produce the most efficient result.

To ensure optimal performance, the optimizer can take advantage of visibility of as much of the application as possible. The SC100 global binder links all modules into a single module on which all optimizations can be performed. As a result of this cross-file approach, the performance of the optimizer is substantially improved, and the generated code is typically more efficient than if produced without cross-file optimization.

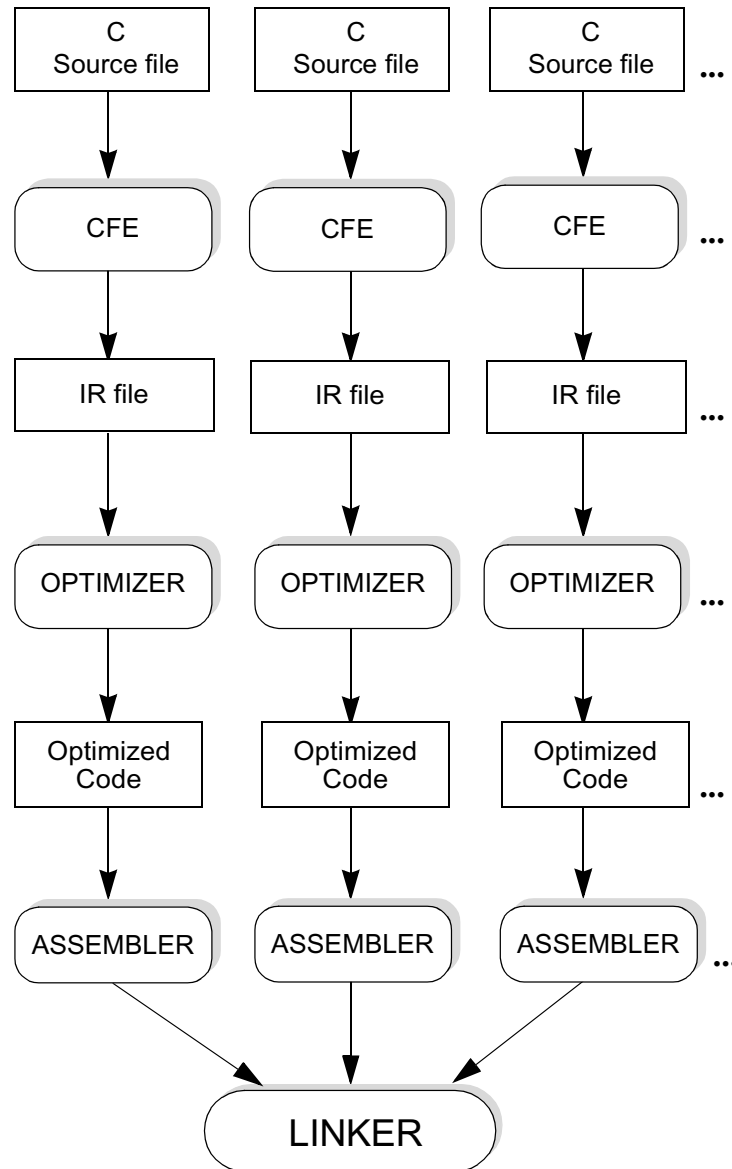
Traditional optimization provides faster compilation, but produces less optimized code. This can be useful during the early stages of development, when you may need to compile different parts of the application separately.

Cross-file optimization produces more efficient code, but the optimization process itself is slower than traditional optimization.

NOTE By default, the shell compiles source files without cross-file optimization, for development purposes. You can choose to specify cross-file optimization when you invoke the shell.

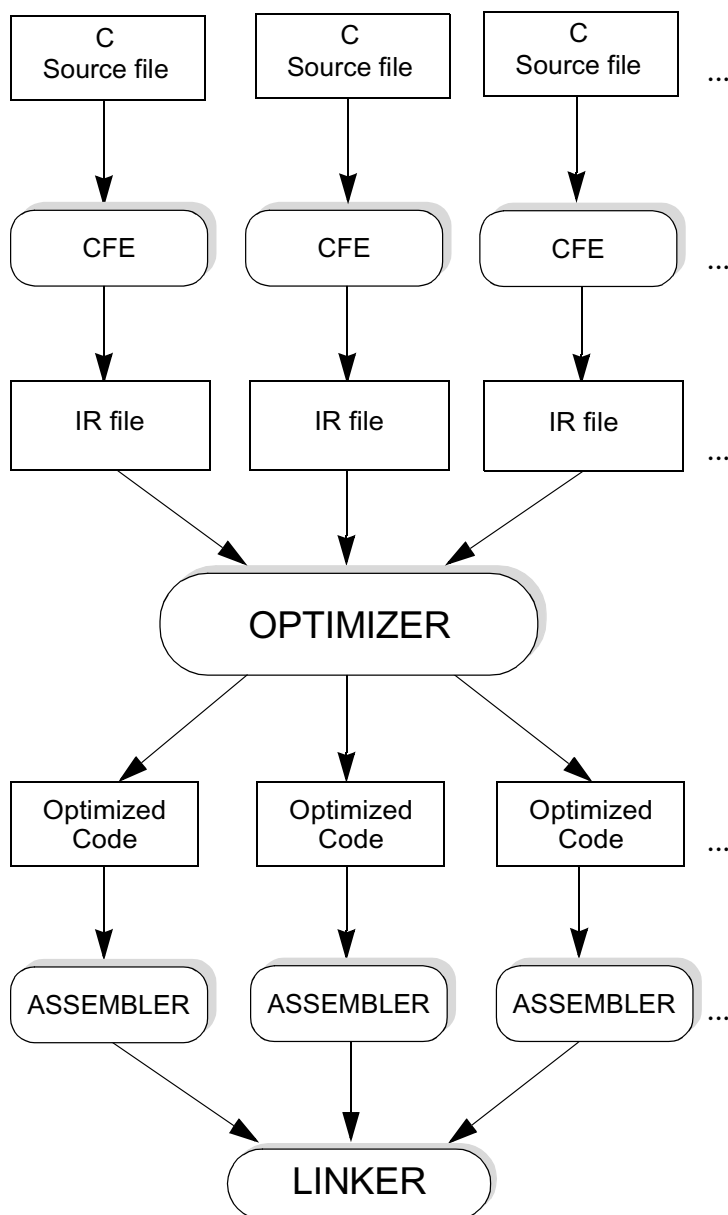
[Figure 3.2](#) shows the processing route for traditional optimization.

Figure 3.2 Traditional optimization



[Figure 3.3](#) shows the processing route for cross-file optimization.

Figure 3.3 Cross-file optimization



File Types and Extensions

The shell program assumes that all items included in the command line that are not recognizable as options or option arguments are input file names. The extension for each file identifies the file type, and determines at which stage the shell will start processing the file. If the file extension is not recognized by any of the tools, the file will be treated as an input file to the linker.

[Table 3.1](#) lists the file extensions and their corresponding file types and shows which tool processes each file type.

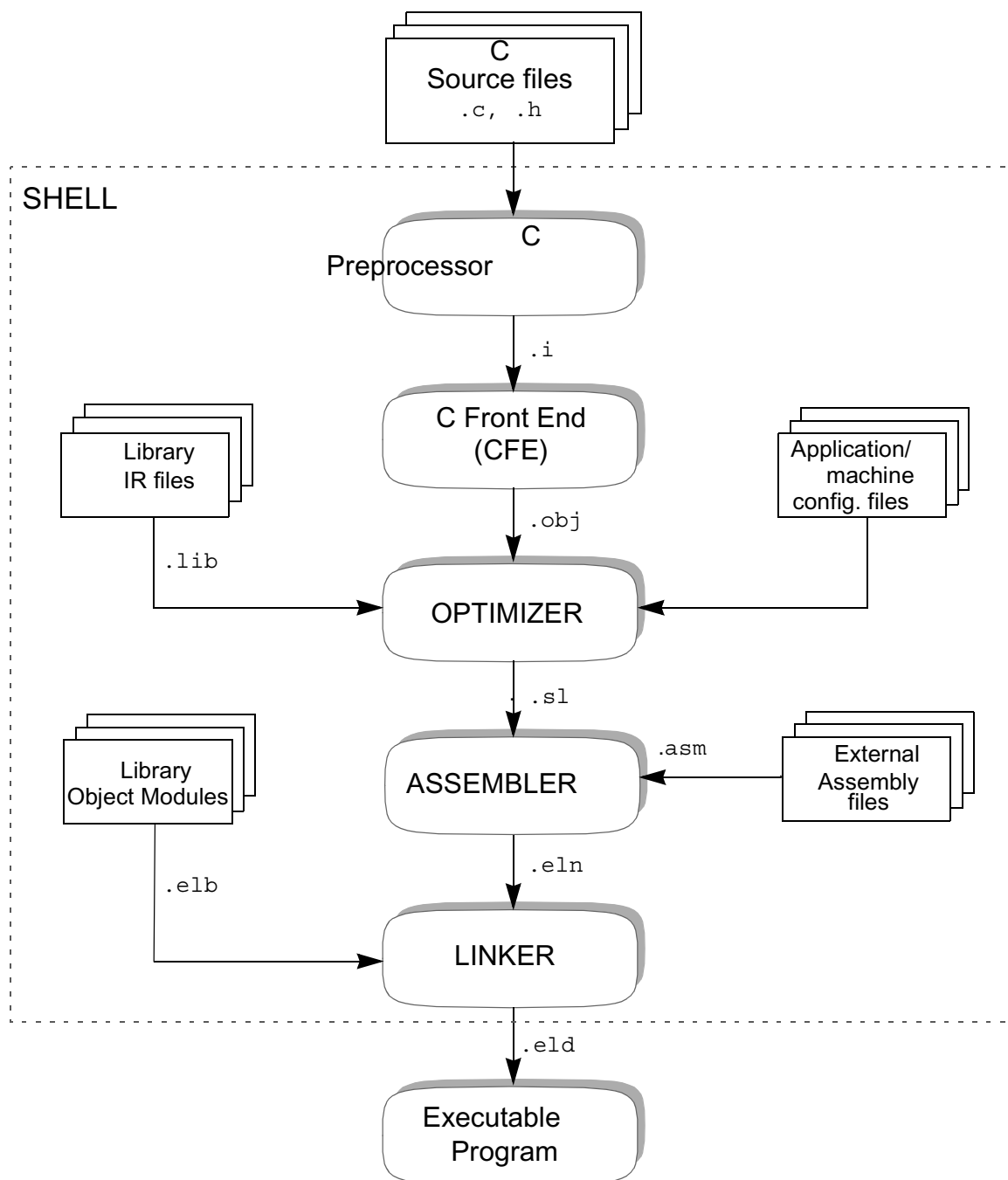
Table 3.1 File types and extensions

Extension	File	Tool
.c	C source file	C Preprocessor
.h	C header file	
.i	Preprocessed C source	Front End
.obj	IR language file	Optimizer
.lib	IR library	Optimizer
.asm, .sl	Assembly file	Assembler
.eln	Relocatable ELF object file	Linker
.cmd, .mem	Linker command file	Linker

NOTE It is possible to cause the shell to process a file as if it were a different file type.

The end result of the compilation process is an executable object file, with a file extension of .eld. [Figure 3.4](#) illustrates the assignment of file extensions at each stage of the shell processing cycle.

Figure 3.4 File extensions in the shell cycle



Environment Variables

Each time the shell executes, it refers to certain environment variables which determine specific aspects of its behavior. These environment variables are defined during the installation process, and include \$SCTOOLS_HOME.

The \$SCTOOLS_HOME environment variable defines the root directory in which the executables, libraries, and tools are stored. This is set to the default location at installation. The compiler searches this directory for all the configuration and executable files that it requires.

Invoking the Shell

The shell is invoked using a single command line, entered at a UNIX® or MS-DOS® prompt. This command line consists of the shell invocation command, one or more file names, and optionally, one or more shell options.

The syntax of the shell command line is as follows:

`scc [option...] file...`

[Table 3.2](#) describes the command line syntax items:

Table 3.2 Command line syntax descriptions

Syntax Item	Description
<i>scc</i>	(Formerly <i>ccsc100</i>). Invokes the compiler shell.
<i>option</i>	One or more options which control the way in which the shell and the various development tools operate. It is not mandatory to specify options in the command line.
<i>file</i>	The names (including extensions) of one or more files to be processed by the shell. These can be source, object, library, and/or command files.

The following syntax rules apply:

- The command line must consist of only one line.
- Individual options and files can be included in the command line in any order, and must be separated from each other by at least one space.

- Options may not be combined, and must be specified individually.
- Options which specify an argument, such as a file name or directory name, must be followed immediately by their argument(s), separated by at least one space.
- All file names, options, and arguments are case sensitive. File names may be any combination of alphanumeric characters and the underscore (_) character.

The shell command line shown in [Listing 3.1](#) specifies three C source files and the option -c, which instructs the shell to compile and assemble these files.

Listing 3.1 Invoking the shell

```
scc -c one.c two.c three.c
```

Using a Command File

The command line can include one or more shell command files. These are files that you can create containing any number of options and arguments, which the shell will use as if they were part of the command line.

Shell Control Options

The options specified in the command line and command files control the operation of the shell, and of the tools used in the application development process.

This section contains these topics:

- [Option Summary](#)
- [Controlling the Behavior of the Shell](#)
- [Specifying Preprocessing Options](#)
- [Overriding Input File Extensions](#)
- [Output Filename and Location Options](#)
- [Specifying C Language Options](#)
- [Passing Options Through to Specific Tools](#)
- [Setting the Options for Listings and Messages](#)

- [Specifying the Hardware Model and Configuration](#)

Option Summary

The following categories of options are provided:

- Options that control the behavior of the shell
- Preprocessing options
- Options that override the file extension for input files
- Output filename and location options
- C language options
- Optimization pragma and code options
- Options that control the output of listing files and messages
- Pass-through options
- Hardware model and configuration options

[Table 3.3](#) provides a summary of the available options.

Table 3.3 Shell options summary

Shell Option	Effect
Options that control the behavior of the shell	
-E [<i>file</i>]	Stops after preprocessing source files. Removes comments.
-cfe	Stops after Front End. Does not invoke the optimizer. Enables the creation of libraries of object files for use with cross-file optimization.
-S	Stops after compilation. Does not invoke the assembler.
-c	Compiles and assembles only. Does not invoke the linker.
-F <i>file</i>	Reads options from the specified file, and appends to command line.
-h or none	Displays the shell Help page, listing all available options.
Preprocessing options	
-C	Preserves comments in the preprocessing output.
-M <i>file</i>	Generates a make file showing dependencies.
-MH <i>file</i>	Generates a list of #include files.
-D <i>mac</i> [=def]	Defines preprocessor macro.
-U <i>macro</i>	Undefines preprocessor macro.
-I <i>dir</i>	Adds directories to the #include file search path.

Using the Metrowerks Enterprise C Compiler

Option Summary

Table 3.3 Shell options summary (*continued*)

Shell Option	Effect
	Syntax note: The options <code>-D</code> , <code>-U</code> , and <code>-I</code> do not require a space before the argument.
Options that override the file extension for input files	
<code>-xc file [file2 ...]</code>	Treats specified file(s) as C source file(s) (<code>.c</code>).
<code>-xobj file [file2 ...]</code>	Treats specified file(s) as IR language file(s) (<code>.obj</code>).
<code>-xasm file [file2 ...]</code>	Treats specified file(s) as assembler source file(s) (<code>.asm</code> or <code>.sl</code>).
Output filename and location options	
<code>-o file</code>	Assigns a filename (and extension) to the output file.
<code>-r dir</code>	Redirects all output to the specified directory.
C language options	
<code>-ansi</code>	Strict ANSI mode. Assumes all C source files contain ANSI/ISO versions of the language, with no extensions. The default mode is the ANSI/ISO version with extensions.
<code>-kr</code>	K&R/pcc mode. Assumes all C source files contain K&R/pcc versions of the language. The default mode is the ANSI/ISO version with extensions.
<code>-g</code>	Adds debug information to generated files.
<code>-ge</code>	Adds DWARF debug extensions to generated files.
<code>-sc</code> (Default)	Makes <code>char</code> type variables signed.
<code>-usc</code>	Makes <code>char</code> type variables unsigned. The default setting is signed.
Optimization pragma and code options	
<code>-O0</code>	Disables all optimizations. Outputs unoptimized assembly code.
<code>-O1</code>	Performs all target-independent optimizations, and outputs optimized linear assembly code. Omits all target-specific optimizations.
<code>-O2</code> (Default)	Performs all optimizations, producing the highest performance code possible without cross-file optimization. Outputs optimized non-linear assembly code.
<code>-O3</code>	Performs the same optimizations as <code>-O2</code> and global register allocation, which results in fewer cycles. (Virtual register allocation is used in this case instead of physical register allocation.)
<code>-Os</code>	Performs space optimization for the indicated level of optimization. Outputs optimized assembly code which is small. This option can be specified together with any of the optimization options except <code>-O0</code> .

Table 3.3 Shell options summary (*continued*)

Shell Option	Effect
-Og	Performs cross-file optimization, which applies the indicated level of optimization across all input files at once. The default is non-cross file optimization. This option can be specified together with any of the optimization options except -O0.
Pass-through options	
-Xasm <i>option</i>	Passes <i>option</i> to the assembler.
-Xlnk <i>option</i>	Passes <i>option</i> to the linker.
Options that control the output of listing files and messages	
-de	Retains a generated error file for each source file.
-dm [<i>file</i>]	Generates a link map file.
-do	Adds to the assembly output file the offsets for C data structure field definitions.
-dL	Generates a C list file for each source file.
-dL1	Generates a C list file for each source file, including a list of #include files.
-dL2	Generates a C list file for each source file, including expansions.
-dL3	Generates a C list file for each source file, including both #include files and expansions.
-dx [<i>file</i>]	Generates a cross-reference information file.
-dc [0-4]	Generates a file showing calls in graphical tree form, in postscript. The number 0 to 4 specifies the paper size, A0 through A4.
-q or -w (Default)	Quiet mode. Displays errors only.
-v	Verbose mode. Displays full information.
-n	Displays command lines without executing.
-s	Causes the compiler to keep assembly language files that it generates (.sl files). (Keeping these files does not stop the shell from performing assembly and linking.)
-Wall	Reports all warnings and remarks.
Hardware model and configuration options	
-arch <i>target</i>	Specifies the target architecture. Valid <i>target</i> values are sc110, sc140e, and sc140 (the default).
-mmac	Specifies the available number of MAC units so that the optimizer can produce parallelized code that fully uses the available number of MAC units. (Valid choices are 1, 2, or 4, depending on your hardware.)

Using the Metrowerks Enterprise C Compiler

Option Summary

Table 3.3 Shell options summary (*continued*)

Shell Option	Effect
-mc <i>file</i>	Specifies the file to be used as the machine configuration file, if different from the default file defined at installation.
-ma <i>file</i>	Specifies the file to be used as the application configuration file, if different from the default file defined at installation.
-crt <i>file</i>	Specifies the file to be used as the startup file, if different from the default file defined at installation.
-mb	Compiles in big-memory mode.
-mrom	Copies all initialized variables from ROM at startup.
-be	Generates output for a big-endian target configuration. The default is a little-endian configuration.
-mem <i>file</i>	Specifies the linker command file to be used, if different from the default file defined at installation.
-mod	Causes the compiler to use modulo addressing.

Controlling the Behavior of the Shell

The options described in this section enable you to control the overall actions of the shell. You can specify the stage at which the shell program will stop processing, define files containing command line options, and display the invocation commands.

Controlling where the shell stops processing

By default, the shell will complete the entire processing cycle, from the input of source files through all the intermediate stages to the output of the final executable. If you want to stop the processing at a specific stage, you can use one of the options `-E`, `-cfe`, `-S`, or `-c`. In this way, you can process and check individual files or groups of files through different stages, until they are finally ready to be compiled and linked together.

Select one of the options described in [Table 3.4](#).

Table 3.4 Options to stop processing in the shell

Option	Description
<code>-E [file]</code>	<p>The shell stops after preprocessing the C source files. Use a <code>.i</code> extension on the file name if the file is to be input to the compiler at a later time. To send output to <code>stdout</code>, do not specify a filename and, in addition, specify the <code>-c</code> option.</p> <p>For example, the following command sends preprocessor output to the file <code>foo.i</code>:</p> <pre>scc -E "foo.i" "foo.c"</pre> <p>This example sends preprocessor output to <code>stdout</code>:</p> <pre>scc -E -c "foo.c"</pre> <p>Comments are not preserved in the preprocessing output, unless the option <code>-C</code> is specified.</p>
<code>-cfe</code>	<p>The shell stops after processing the input source files through the Front End. You can use this option to check that the files are valid source files, which meet the essential requirements for processing by the shell, for example, they contain no syntax errors. This is primarily useful when preparing files for cross-file optimization. Output files are IR files, assigned the extension <code>.obj</code>. The <code>-cfe</code> option enables you to create libraries of object files for use later when compiling in cross-file optimization mode.</p>

Table 3.4 Options to stop processing in the shell

Option	Description
-S	The shell stops after compiling the source files to assembly files, and does not invoke the assembler. Output files are assigned the extension <code>.sl</code> .
-c	The shell stops after compiling C and assembly source files to object code, and does not invoke the linker. The object code output files are assigned the extension <code>.eln</code> .

Following processing with any of the above options, the output files are written to the current directory, or if the `-r` option has been included, to the specified directory. The output files are assigned the same names as the input files, with the extension for the selected option, as shown above. Any existing files in the directory with the same name and extension are overwritten.

The starting point for the processing of each input file is determined by its file extension.

Specifying a shell command file

You can create command files containing options and arguments, which the shell program will treat as if they were included on the command line.

Defining options and arguments within command files can save you input time when you invoke the shell program, and helps you overcome any imposed limitation on the length of the command line. Each time you invoke the shell, you can select the command file with the set of options that suit your specific requirements.

To specify a shell command file, specify the option `-F` followed by a filename. A command file can itself contain the option `-F` specifying another shell command file.

[Listing 3.2](#) illustrates the use of the `-F` option to specify the command file `proj.opt`.

Listing 3.2 Defining a shell command file

```
scc -F proj.opt
```

Within the command file, each separate option (with or without an argument), file, or list of files must reside on a new line. You can specify as many lines as you wish, in any order. Comments can be

included in the file using the # character. All characters between # and the end of the line are ignored by the shell.

The command file shown in [Listing 3.3](#) contains four lines which instruct the shell to invoke the linker with three application object files and one library file, generate a link map file, and output the executable program to a file named `appl.eld`.

Listing 3.3 Contents of a shell command file

<code>-o appl.eld</code>	<code># output file name</code>
<code>-dm appl.map</code>	<code># generate map file</code>
<code>file1.eln file2.eln file3.eln</code>	<code># object files</code>
<code>-l mylib.elb</code>	<code># shared library</code>

NOTE If no map file is specified, the shell generates a file with the same file name as the specified `.eld` file, and the extension `.map`.

Displaying the shell Help page

You can display the shell Help page, which takes the form of a list of all the available shell options and arguments. Select the option `-h` to display this list.

[Listing 3.4](#) shows a section of the shell Help page:

Listing 3.4 Shell Help page (extract)

<code>-c</code>	Compile and assemble only. Don't invoke the linker
<code>-cfe</code>	Stop after Front-End. (Used for cross-file optimization)
<code>-S</code>	Generate assembly output file. Don't invoke assembler
<code>-E</code>	Preprocess only
<code>-C</code>	Preprocess only and keep comments

Specifying Preprocessing Options

The options described in this section enable you to control the preprocessing stage of the shell program, before the input files proceed through the Front End. You can change the output produced by the preprocessor, define one or more preprocessor macros, and define the directories to be searched for `#include` files.

Changing preprocessed output

You can specify any of the options in [Table 3.5](#) to change the format and content of the preprocessed output. These options can be specified in addition to the `-E` option, or instead of the `-E` option.

Table 3.5 Options to change preprocessed output

Option	Description
<code>-C</code>	Keeps all comments (preprocessor directives) in the preprocessing output. If you specify the <code>-E</code> option only, the preprocessed text is written to the output file with line control information only, and with all comments removed.
<code>-M [file]</code>	Instead of the normal preprocessing output, an output file is generated in MAKE format, containing a list showing the dependencies between the input source files. If no file is specified, the output is sent to the standard output stream, <code>stdout</code> .
<code>-MH [file]</code>	Instead of the normal preprocessing output, an output file is generated containing a list of all the <code>#include</code> files used in the source. This list includes all levels of <code>#include</code> files, together with any nested files. If no file is specified, the output is sent to the standard output stream, <code>stdout</code> .

Defining and undefining preprocessor macros

You can define one or more preprocessor macros, and you can remove the definition of a macro.

You can specify the macro options in [Table 3.6](#) more than once in the command line, to define and undefine different preprocessor macros.

Table 3.6 Macro options

Option	Description
<code>-D macro [=value]</code>	<p>Defines the named macro as a preprocessor macro, with the specified value. If <code>value</code> is omitted, the value 1 (one) is assumed. Once a preprocessor macro is defined with this option, it is passed by the shell to the preprocessor for all subsequent compilations until it is undefined with the <code>-U</code> option.</p> <p>The space between the <code>-D</code> option and the named macro is optional.</p>
<code>-U macro</code>	<p>Undefines the named macro by removing its previous definition. The macro will not be passed to the preprocessor unless it is redefined with the <code>-D</code> option. Any <code>-U</code> options in the command line are processed only after all <code>-D</code> options have been processed.</p> <p>It is not necessary to enter a space between the <code>-U</code> option and the named macro.</p>

Adding directories to the #include file path

The option `-I dir` adds the specified directory or directories to the path used by the shell to search for `#include` files. The string *dir* can be a list of directories.

To specify directory or directories for the `#include` file search path, specify the option `-I`, followed by a directory name or a list of directories. The space between the `-I` option and the *dir* string is optional. On UNIX hosts, separate the individual directories in the list with colons (:). On PC hosts, separate the individual directories with semicolons (;).

You can use this option more than once in a command line, and the directories or lists will be searched in the order in which the options are supplied.

Overriding Input File Extensions

You can change how the shell program treats a specific input file, by overriding the assumptions made by the shell based on the file's extension.

You can select any of the options in [Table 3.7](#), as many times as required. After the selected option you can specify one or more filenames, separated by spaces.

Table 3.7 Options to override input file extensions

Option	Description
<code>-xc <i>file</i> [<i>file2</i> ...]</code>	This option identifies the specified files as C language source files, as if they had the extension <code>.c</code> . The shell will process these files in exactly the same way as any other C source files specified in the command line, subject to any other processing options selected.
<code>-xobj <i>file</i> [<i>file2</i> ...]</code>	This option identifies the files as IR language files, as if they had been output by the Front End with the extension <code>.obj</code> . The files will be input for processing by the compiler.
<code>-xasm <i>file</i> [<i>file2</i> ...]</code>	This option instructs the shell program to identify the specified files as assembler source files, as if they had the extension <code>.asm</code> or <code>.sl</code> . The files will be assembled at the appropriate processing stage, and the object code will be made available to the linker.

These options can appear any number of times in the command line. Each option relates to one specified file or a list of files. The files

identified by these options are processed normally in all other respects, and in the same relative order as other listed files.

In [Listing 3.5](#), the input files `file1.ext` and `file2.bar`, specified after the option `-xc`, will be compiled as if they were C source files.

Listing 3.5 Overriding file extensions

```
scc -c -xc file1.ext file2.bar
```

Output Filename and Location Options

These options let you specify the name and/or directory for the output files which the shell program will produce. By default, each output file is assigned the same name as the input file and is stored in the current directory.

The default file type and extension for the output files are determined by the stage at which the shell stops processing. For example, when the `-cfe` option has been selected, the output files produced by the Front End will have the extension `.obj`. If you wish, you can specify a different extension when you specify the file name. This will alter the way the shell will treat this file.

You can select either or both of the options in [Table 3.8](#).

Table 3.8 Output file name and location options

Option	Description
<code>-o file</code>	The output file is assigned the specified filename, and optionally the specified extension. Any existing file with the same name in the current directory, or in the specified directory, if the <code>-r</code> option is selected, is overwritten. You can specify this option more than once in the command line, for different files.
<code>-r dir</code>	All output files are redirected to the specified directory. This option can be specified only once in the command line.

In [Listing 3.6](#), the input file `file1.foo` will be treated as an input file to the linker (the default).

Listing 3.6 Specifying output files

```
scc -o file.eld file1.foo
```

Specifying C Language Options

You can use the C language options described in this section to inform the shell of the language version being used in the source files, to add debugging information to generated files, and to define whether variables of type `char` should default to signed or unsigned.

Defining the language version

The default C language mode is the normal ANSI/ISO version with extensions, with all source files using the standard `.c` extension. You do not need to specify any language option if you use this mode. If, however, you use a different language version, you must select either the `-ansi` or the `-kr` option.

If you use the strict ANSI/ISO version of C, select the option `-ansi`. All input source files will be assumed by the Front End to be in the strict ANSI/ISO version of C, with no extensions. Any extensions found will be flagged with warnings.

If you use the K&R (Portable C Compiler, or PCC) dialect of C, select the option `-kr`. The shell program will assume that all source files are in this version of C.

You cannot compile source files in different C language versions at the same time. If you need to compile source files in different versions, you must use a separate shell command line for each version.

Adding debugging information to files

The option `-g` causes the shell program to include debugging information in the output files produced by all C compilations. The object files that are produced will be somewhat larger as they will contain source-level debugging information.

Optimization is disabled, by default, when the `-g` option is specified. This default optimization setting is recommended for use with debugging. If you specify an optimization level other than `-O0` in combination with `-g`, the following warning message is issued:

"Warning: Debugging with optimized code."

Changing the default char sign setting

The default setting for all char type variables is signed. You can change this setting to make all char type variables default to unsigned using the -usc option. To change the setting back to make all char type variables default to signed, specify the -sc option.

Passing Options Through to Specific Tools

The options described in this section enable you to instruct the shell program to pass options to specific tools, such as the assembler or linker, as shown in [Listing 3.7](#).

Listing 3.7 Passing multiple options to the same tool

```
-Xasm -occ
```

You can specify more than one option to be passed to the same tool in the same option statement, together with the arguments for each option. Multiple options, and their arguments where relevant, must be listed within quotation marks.

If a tool is invoked several times, the pass-through options are passed to the tool on each invocation, in addition to any other options passed directly by the shell program to the tool from the command line.

Specify any of the options in [Table 3.9](#).

Table 3.9 Options used to pass options to specific tools

Option	Description
-Xasm <i>option</i>	Passes the specified options and arguments to the assembler.
-Xlnk <i>option</i>	Passes the specified options and arguments to the linker. For example: scc -Xlnk "-map "foo.map" " foo.eln
-Xllt <i>option</i>	Passes the specified options and arguments to the Low-Level Transform.
-Xcfe <i>option</i>	Passes the specified options and arguments to the Front End.
-Xicode <i>option</i>	Passes the specified options and arguments to ICODE.

NOTE Use the -mem option to pass a command file other than the default to the linker. If you use the -Xlnk option to do this, both the command

file you are specifying and the default command file are passed to the linker, resulting in errors.

Setting the Options for Listings and Messages

The options in this section enable you to control the retention, display and printing of diagnostic and informational messages, and the generation of various listing and map files.

Generating listing files

By default the shell program does not retain the diagnostic and cross-reference information produced at different processing stages. You can select to retain one or more different types of information in listing files.

Use any combination of the options in [Table 3.10](#) to generate listing files containing the types of information you require. Each individual option can only be specified once in a shell command line.

Table 3.10 Options to generate message listing files

Option	Description
-de	The Front End creates a file containing all error messages generated during the compilation. The <code>-de</code> option retains this error file. If this option is not specified, the errors are displayed during processing, but not kept. An error file is created for each source file, with the same name as the source file and the extension <code>.err</code> .
-dm [<i>file</i>]	Generates a link map file listing all the specific variables, applications and addresses used by the linker. If no file name is specified, a file is created with the same name as the executable, and the extension <code>.map</code> .
-do	Includes the details of C data structures in the output assembly file, showing the offsets for all field definitions in each data structure.
-dL	Generates a C list file for each source file, listing the entire contents of the source file. Each list file is created with the same name as its corresponding source file, and the extension <code>.lis</code> .
-dLl	Generates a C list file for each source file, listing the entire contents of the source file, with the addition of a list of <code>#include</code> files used by the source. Each list file is created with the same name as its corresponding source file, and the extension <code>.lis</code> .

Table 3.10 Options to generate message listing files

Option	Description
-dL2	Generates a C list file for each source file, listing the entire contents of the source file, with the addition of expansions, such as macro expansions, line splices, and trigraphs. Each list file is created with the same name as its corresponding source file, and the extension <code>.lis</code> .
-dL3	Generates a C list file for each source file, listing the entire contents of the source file, with the addition of a list of <code>#include</code> files, and expansions, such as macro expansions, line splices, and trigraphs. Each list file is created with the same name as its corresponding source file, and the extension <code>.lis</code> .
-dx [<i>file</i>]	Generates a cross-reference information file, providing details of cross-references in the source file. If no file name is specified, a file is created with the same name as the source file, and the extension <code>.xrf</code> .
-dc [0-4]	Generates a file showing calls in graphical tree form, which can be printed using a postscript printer. Specify the size of the paper to be used for the printout: 0 for paper size A0, 1 for A1, and so on.

Controlling the type of information displayed

You can control the level and type of messages and information that the shell program displays using the options in [Table 3.11](#).

Table 3.11 Options to control information displayed by the shell

Option	Description
-q or -w	Quiet mode (the default). The shell program displays the minimum amount of information (errors only). Normal notices and banners are omitted. This option is useful when running the shell in batch mode or with the <code>MAKE</code> utility, when the display of normal progress information is not required.
-v	Verbose mode. The shell program displays/prints all the commands and command line options being used, as it proceeds through the different processing stages and invokes the individual tools. The exact information output will depend on the processing stages performed by the shell.
-n	Displays the specified shell processing actions without executing them. You can use this option before you invoke the shell, to check the actions the shell will take, based on the list of files and arguments specified in the command line.

Reporting all remarks and warnings

The shell reports all errors and warnings by default, but will not report remarks unless you specifically instruct it to do so. Select the

option `-Wall` to ensure that all remarks are reported, as well as all warnings and errors.

Specifying the Hardware Model and Configuration

The options in this category let you override some of the hardware and configuration settings defined as the default during installation.

Defining the architecture

The default architecture is SC140, which utilizes four MAC units. Unless instructed otherwise, the compiler assumes during the optimization phase that four execution units are in use, and parallelizes the code accordingly.

If you are compiling for a hardware configuration other than SC140, it is essential that you specify the correct architecture. To change the assumed architecture, specify the `-arch target` option, as illustrated in [Listing 3.8](#).

Valid values for *target* are `sc110` and `sc140` (default).

Listing 3.8 Defining the architecture

```
scc -arch sc110 file1.c
```

Defining specific configuration and startup files

The default machine and application configuration files used by the compiler, and the startup file used by the linker, are defined during the installation process.

The machine configuration file includes information about the logical and physical memory maps. This information enables the global optimizer to dispatch variables to different memory areas in internal ROM or RAM.

The application configuration file contains information about how the application software and the hardware interact. The file includes sections about binding interrupt handlers, overlays, and application objects to specific addresses.

The startup file is used by the linker when it links the assembly code files with the standard libraries, and defines such items as the

interrupt vector and set-up code executed upon system initialization.

You may wish to select other files to be used for configuration setup and initialization instead of the default files, for example, to specify certain devices which need to be initialized at startup.

To specify different files to be used at initialization, select one or all of the options in [Table 3.12](#). For each option, specify the file name, and if the file is not in the current directory, specify the path.

Table 3.12 Options to specify configuration and startup files

Option	Description
-mc <i>file</i>	The compiler reads the specified file instead of the default machine configuration file.
-ma <i>file</i>	The compiler reads the specified file instead of the default application configuration file.
-crt <i>file</i>	The linker links into the application the specified file instead of the default startup file.
-mem <i>file</i>	The linker uses the specified command file instead of the default linker command file (<code>crtscsmm.cmd</code> or <code>crtscbmm.cmd</code>).

Defining memory mode

The SC100 architecture instruction set supports both 16-bit and 32-bit addresses. If the application is small enough to allow all static data to fit into the lower 64K of the address space, then more efficient code can be generated. This mode (small memory mode) is the default, and assumes that all addresses are 16-bit.

If your application does not fit into 64K bytes, meaning that the use of 32-bit absolute addresses is required, you must instruct the shell to use the big memory model, by specifying the `-mb` option.

Copying initialized variables from ROM

During development you would normally use a loader to set the values for global variables, and to load these initialized variables into RAM at startup, together with the executable application.

When you have finished development, if your final application does not use a loader, you must ensure that when the completed application executes, the initialized variables will be copied from

ROM into RAM. To do this, when you compile the final application version, specify the `-mrom` option.

Specifying big-endian mode

By default, the compiler generates code based on the assumption that the architecture operates in little-endian mode, meaning the least significant bits in the lower address. If you want to run the application in an environment that operates in big-endian mode, meaning the most significant bits in the lower address, specify the option `-be`.

Language Features

This section describes the different language modes accepted by the Metrowerks Enterprise C compiler. It also provides detailed information about the data types and sizes supported, fractional arithmetic representation, intrinsic functions, pragmas and predefined macros.

The topics in this section are:

- [C Language Dialects](#)
- [Types and Sizes](#)
- [Fractional and Integer Arithmetic](#)
- [Intrinsic Functions](#)
- [Pragmas](#)
- [Predefined Macros](#)

C Language Dialects

The compiler accepts three variations of the C language. The standard C language mode is the normal ANSI/ISO version with extensions. This is the default mode.

[Table 3.13](#) lists the other two accepted C language modes.

Table 3.13 Additional C language modes

C Language Mode	Description
Strict ANSI/ISO mode	Specified with the shell option <code>-ansi</code> . Any ISO C extensions are flagged with warnings.
K&R/PCC mode	Specified with the shell option <code>-kr</code> . The compiler accepts the older K&R dialect of C, and provides almost complete compatibility with the widely used UNIX PCC (<code>pcc</code>) dialect.

Source files of different C language types may not be compiled together, but once compiled they can be linked together into a single application.

Standard Extensions

This section lists the extensions which are normally accepted in standard C programs. When compiling in strict ANSI/ISO mode, the compiler issues warnings when these extensions are used.

Preprocessor extensions

The following preprocessor extensions are accepted:

- Comment text can appear at the end of preprocessing directives.
- Numbers are scanned according to the syntax for numbers. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token.
- The `#assert` preprocessing extensions of AT&T System V release 4 are allowed. These enable the definition and testing of predicate names. Such names are in a name space distinct from all other names, including macro names. A predicate name can be defined by a preprocessing directive in one of two forms, as shown in [Listing 3.9](#):

Listing 3.9 Defining a predicate name

```
#assert name  
#assert name (token-sequence)
```

In the first form, the predicate is not given a value. In the second form, it is given the value *token-sequence*. Such a predicate can be tested in a `#if` expression, as follows:
`#name (token-sequence)`. This expression has the value 1 if a `#assert` of that *name* with that *token-sequence* has

appeared, otherwise it has the value 0. A predicate may be assigned more than one value at a given time.

- A predicate may be deleted by a preprocessing directive in one of two forms, as shown in [Listing 3.10](#):

Listing 3.10 Deleting a predicate

```
#unassert name  
#unassert name(token-sequence)
```

The first form removes all definitions of the indicated predicate name. The second form removes only the indicated definition, leaving any remaining definitions unchanged.

A number of predefined preprocessor macros are provided.

The pragmas supported by the compiler are available in all modes.

Syntax

The following syntax extensions are accepted:

- A translation unit (input file) can be empty, containing no declarations.
- An extra comma is allowed at the end of an enum list. Similarly, the final semicolon preceding the closing `}` of a struct or union specifier may be omitted. A remark is issued in both cases, except in `pcc` mode.
- A label definition may be followed immediately by a right brace. (Normally, a label definition must be followed by a statement.) A warning is issued.
- An empty declaration (a semicolon with nothing before it) is allowed. A remark is issued.
- An initializer expression that is a single value and is used to initialize an entire static array, struct, or union need not be enclosed in braces, except in strict ANSI C mode.
- A struct that has no named fields, but which has at least one unnamed field, is accepted by default. A diagnostic (a warning or error) is issued in strict ANSI C mode.

Declarations

The following declaration extensions are accepted:

- Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.
- Benign redeclarations of `typedef` names are allowed, meaning that a `typedef` name may be redeclared in the same scope as the same type. A warning is issued.
- The compiler always accepts `asm` statements and declarations, with one exception, which is when compiling in strict ANSI C mode. The reason for this is that there would be a conflict with the ANSI C standard. For example, `asm("xyz");` would be interpreted by the Front End as an `asm` statement by default, while ANSI C would interpret this as a call of an implicitly-defined function `asm`.
- Functions declared as `asm` functions are accepted, and `__asm` is recognized as a synonym for `asm`. An `asm` function body is represented by an uninterpreted null-terminated string containing the text that appears in the source.
- An `asm` function must be declared with no storage class, with a prototyped parameter list, and with no omitted parameters, as shown in [Listing 3.11](#):

Listing 3.11 Declaring an `asm` function

```
asm void f(int,int) {  
    ...  
}
```

- As an `asm` function must be output with a prototyped parameter list, these functions are valid for ANSI C modes only.

Types

The following type extensions are accepted:

- Bit-fields may have base types that are enums or integer types, as well as the types `int` and `unsigned int`. The use of any signed integer type is equivalent to using type `int`, and the use of any unsigned integer type is equivalent to using type `unsigned int`.
- The last member of a `struct` may have an incomplete array type. It may not be the only member of the `struct` (otherwise, the `struct` would have zero size).
- A file-scope array may have an incomplete `struct`, union, or enum type as its element type. The type must be completed

before the array is subscripted (if it is subscripted), and by the end of the compilation if the array is not `extern`.

- The enum tags may be incomplete. The tag name can be defined and resolved later (by specifying the brace-enclosed list).
- Object pointer types and function parameter arrays that decay to pointers may use `restrict` as a type qualifier. Its presence is recorded in the compiler so that optimizations can be performed that would otherwise be prevented because of possible aliasing.
- The type `long float` is accepted as a synonym for `double`.
- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `const int **`).

Expressions and statements

The following extensions are accepted for expressions and statements:

- Assignment and pointer differences are allowed between pointers to types that are interchangeable, but not identical, for example, `unsigned char *` and `char *`. This includes pointers to same-sized integral types (e.g., typically, `int *` and `long *`). A warning is issued, except in `pcc` mode. A string constant may be assigned to a pointer to any kind of character, without a warning.
- In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ANSI C, some operators allow such conversions, while others do not, generally where such a conversion would not be logical.
- In an initializer, a pointer constant value may be cast to an integral type if the integral type is big enough to contain it.
- In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- In character and string escapes, if the character following the `\` has no special meaning, the value of the escape is the character itself. Thus `"\s" == "s"`. A warning is issued.
- Adjacent wide and non-wide string literals are not concatenated.
- In duplicate size and sign specifiers (e.g., `short short` or `unsigned unsigned`) the redundancy is ignored, and a warning is issued.

- `__ALIGNOF__` is similar to `sizeof`, but returns the alignment requirement value for a type, or 1 if there is no alignment requirement. It may be followed by a type or expression in parentheses, as shown in [Listing 3.12](#):

Listing 3.12 Returning the alignment requirement

```
__ALIGNOF__ (type)
__ALIGNOF__ (expression)
```

The expression in the second form is not evaluated.

- Identifiers may not contain dollar signs.
- `__INTADDR__ (expression)` scans the enclosed expression as a constant expression, and converts it to an integer constant (it is used in the `offsetof` macro).
- The values of enumeration constants may be given by expressions that evaluate to unsigned quantities which fit in the unsigned int range but not in the int range. A warning is issued when such a result is possible, as shown in [Listing 3.13](#):

Listing 3.13 Out of range warning

```
/* When ints are 32 bits: */
enum a {w = -2147483648}; /* No warning */
enum b {x = 0x80000000}; /* No warning */
enum c {y = 0x80000001}; /* No warning */
enum d {z = 2147483649}; /* Warning */
```

- The address of a variable with register storage class may be taken, and a warning is issued.
- The expression `& . . .` is accepted in the body of a function in which an ellipsis appears in the parameter list.
- An ellipsis may appear by itself in the parameter list of a function declaration, for example, `f (. . .)`. A diagnostic is issued in strict ANSI mode.
- External entities declared in other scopes are visible, as shown in [Listing 3.14](#). A warning is issued.

Listing 3.14 External entities in other scopes

```
void f1(void) { extern void f(); }  
void f2() { f(); /* Using out of scope declaration */ }
```

- Pointers to incomplete arrays may be used in pointer addition, subtraction, and subscripting, as shown in [Listing 3.15](#).

Listing 3.15 Pointers to incomplete arrays

```
int (*p)[];  
...  
q = p[0];
```

A warning is issued if the value added or subtracted is anything other than a constant zero. Since the type pointed to by the pointer has zero size, the value added to or subtracted from the pointer is multiplied by zero and therefore has no effect on the result. Comparisons and pointer differences of such pairs of pointer types are also allowed. A warning is issued.

- Pointers to different function types may be assigned or compared for equality (==) or inequality (!=) without an explicit type cast, and a warning is issued.
- A pointer to void may be implicitly converted to or from a pointer to a function type.
- Intrinsic functions are recognized as extensions only in the default C language mode (ANSI C with extensions). In all other modes they are treated as function calls.

K&R/PCC mode

When pcc mode is specified, the Metrowerks Enterprise C compiler accepts the traditional C language defined by *The C Programming Language*, first edition, by Kernighan and Ritchie (K&R), Prentice-Hall, 1978. This mode provides almost complete compatibility with the Reiser CPP and Johnson PCC (pcc), both widely used as part of UNIX systems. Since there is no documentation of the exact behavior of those programs, complete compatibility cannot be guaranteed.

In general, when compiling in pcc mode, the compiler attempts to interpret a source program that is valid to pcc in the same way that

pcc would. However, ANSI features that do not conflict with this behavior are not disabled.

In some cases where pcc allows a highly questionable construct, the compiler accepts it but gives a warning, where pcc would be silent. For example: 0x, a degenerate hexadecimal number, is accepted as zero, but a warning is issued.

K&R/PCC mode preprocessor differences

The following are the preprocessor differences relative to the default standard mode:

- When preprocessing output is generated, the line-identifying directives will have the pcc form instead of the ANSI form.
- `__STDC__` is left undefined.
- Comments are deleted entirely (instead of being replaced by one space) in preprocessing output. Extra spaces are not generated in textual preprocessing output to prevent pasting of adjacent confusable tokens. As a result, the characters `a/**/b` will be `ab` in preprocessor output.
- The first directory searched for include files is the directory containing the file which contains the `#include` instead of the directory which contains the primary source file.
- Trigraphs are not recognized.
- Macro expansion is implemented differently. Arguments to macros are not macro-expanded before being inserted into the expansion of the macro. Any macro invocations in the argument text are expanded when the macro expansion is rescanned. With this method, macro recursion is possible and is checked for.
- Token pasting inside macro expansions is implemented differently. End-of-token markers are not maintained, so tokens that abut after macro substitution may be parsed as a single token.
- Macro parameter names inside character and string constants are recognized and are given substitutes.
- Macro invocations having too many arguments are flagged with a warning rather than an error. The extra arguments are ignored.
- Macro invocations having too few arguments are flagged with a warning rather than an error. A null string is used as the value of the missing parameters.

- Extra occurrences of `#else` (after the first has appeared in an `#if` block) are ignored, with a warning.

K&R/PCC mode syntax differences

The following are the syntax differences relative to the default standard mode:

- The keywords `signed`, `const`, and `volatile` are disabled, so that they can be user identifiers. The other non-K&R keywords (`enum` and `void`) are judged to have existed already in code and are not disabled.
- The `=` preceding an initializer may be omitted. A warning is issued. This was an anachronism even in K&R.
- `0x` is accepted as a hexadecimal 0, with a warning.
- `1E+` is accepted as a floating point constant with an exponent of 0, with a warning.
- The compound assignment operators may be written as two tokens (for example, `+=` may be written `+ =`).
- The compound assignment operators may be written in their old-fashioned reversed forms (for example, `-=` may be written `= -`). A warning is issued.
- The digits 8 and 9 are allowed in octal constants. (For example, the constant `099` has the value $9*8+9$, or 81.)
- The escape `\a` (alert) is not recognized in character and string constants.

K&R/PCC mode differences for declarations

The following are the declaration differences relative to the default ANSI mode:

- Declarations of the form `typedef some-type void;` are ignored.
- The names of functions and of external variables are always entered at the file scope.
- A function declared `static`, which is used and never defined, is treated as if its storage class were `extern` (instead of causing an error for being undefined).
- A file-scope array that has an unspecified storage class and remains incomplete at the end of the compilation will be treated as if its storage class is `extern`. In ANSI mode, the number of

elements is changed to 1, and the storage class remains unspecified.

- When a function parameter list begins with a typedef identifier, the parameter list is considered prototyped only if the typedef identifier is followed by something other than a comma or right parenthesis, as shown in [Listing 3.16](#).

Listing 3.16 Prototyped parameter list

```
typedef int t;  
int f(t) {}      /* Old-style list */  
int g(t x) {}    /* Prototyped list, parameter x of type t */
```

Function parameters are allowed to have the same names as typedef identifiers. In the normal ANSI mode, any parameter list that begins with a typedef identifier is considered prototyped, and [Listing 3.16](#) would produce an error.

- The empty declaration `struct x;` will not hide an outer-scope declaration of the same tag. It is taken to refer to the outer declaration.
- In a declaration of a member of a struct or union, the declarator list may be omitted entirely, to specify an unnamed field which requires padding, as shown in [Listing 3.17](#). Such a field may not be a bit-field.

Listing 3.17 Omitting the declarator list

```
struct s {char a; int; char b[2];} v; /* sizeof(v) is 3 */
```

- No warning is generated for a storage specifier appearing in other than the first position in a list of specifiers (as in `int static`).
- The keywords `short`, `long`, and `unsigned` are treated as “adjectives” in type specifiers, and they may be used to modify a typedef type. For example, the declarations in [Listing 3.18](#) result in `s` having type `unsigned long`:

Listing 3.18 Keywords in type specifiers

```
typedef long size;  
unsigned size s;
```

- Free-standing tag declarations are allowed in the parameter declaration list for a function with old-style parameters.
- Declaration specifiers are allowed to be completely omitted in declarations. (ANSI C allows this only for function declarations.) Thus `i;` declares `i` as an `int` variable. A warning is issued.
- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.

K&R/PCC mode type differences

The following are the type differences relative to the default standard mode:

- Integral types with the same representation (size, signedness, and alignment) will be considered identical and may be used interchangeably. For example, this means that `int` and `long` will be interchangeable if they have the same size.
- All enums are given type `int`. In ANSI mode, smaller integral types will be used if possible.
- A “plain” `char` is considered to be the same as either `signed char` or `unsigned char`, depending on the command-line options. In ANSI C, “plain” `char` is a third type distinct from both `signed char` and `unsigned char`.
- All `float` functions are promoted to `double` functions, and any `float` function parameters are promoted to `double` function parameters.
- All `float` operations are executed as `double`.
- The types of large integer constants are determined according to the K&R rules. They will not be unsigned in some cases where ANSI C would define them that way.

K&R/PCC mode differences: expressions and statements

The following are the differences for expressions and statements relative to the default standard mode:

- Assignment is allowed between pointers and integers, and between incompatible pointer types, without an explicit cast. A warning is issued.

- A field selection of the form `p->field` is allowed even if `p` does not point to a `struct` or `union` that contains `field`. In this context, `p` must be a pointer or an integer. Similarly, `x.field` is allowed even if `x` is not a `struct` or `union` that contains `field`. In this case, `x` must be an lvalue. In both cases, if `field` is declared as a `field` in more than one `struct` or `union`, it must have the same offset in all instances.
- Overflows detected while folding signed integer operations on constants will cause warnings rather than errors.
- A warning will be issued for an `&` operator applied to an array. The type of such an operation is “address of array element” rather than “address of array”.
- For the shift operators `<<` and `>>`, the usual arithmetic conversions are done on the operands as they would be for other binary operators. The right operand is then converted to `int`, and the result type is the type of the left operand. In ANSI C, the integral promotions are done on the two operands separately, and the result type is the type of the left operand. The effect of this difference is that, in `pcc` mode, a long shift count will force the shift to be done as `long`.
- String literals will not be shared. Identical string literals will cause multiple copies of the string to be allocated.
- The expression `sizeof` may be applied to bit-fields. The size is that of the underlying type (for example `unsigned int`).
- Any lvalues cast to a type of the same size remain lvalues, except when they involve a floating point conversion.
- A warning rather than an error is issued for integer constants that are larger than can be accommodated in an `unsigned long`. The value is truncated to an acceptable number of low-order bits.
- Expressions in a `switch` statement are cast to `int`. This differs from the ANSI C definition in that a `long` expression may be truncated.
- The promotion rules for integers are different: `unsigned char` and `unsigned short` are promoted to `unsigned int`.

K&R/PCC differences: remaining incompatibilities

The additional known cases where the compiler is not compatible with `pcc` are as follows:

- Token pasting is not implemented outside of macro expansions (meaning, in the primary source line) when two tokens are separated only by a comment. That is, `a/**/b` is not considered to be `ab`. The `pcc` compiler's behavior in such a case can be obtained by preprocessing to a text file and then compiling that file.

The textual output from preprocessing is also equivalent but not identical. The blank lines and white space will not be exactly the same as those produced in `pcc`.

- The `pcc` compiler considers the result of a `? :` operator to be an `lvalue` if the first operand is constant and the second and third operands are compatible `lvalues`. The compiler never treats the result of the `? :` operator as an `lvalue`.
- The `pcc` compiler misparses the third operand of a `? :` operator in a way that some programs exploit, as follows:

`i ? j : k += 1` is parsed by `pcc` as `i ? j : (k += 1)`

This is not correct, since the precedence of the `+=` operator is lower than the precedence of the `? :` operator. The compiler will generate an error in such a case.

- The `lint` utility recognizes the keywords for its special comments anywhere in a comment, regardless of whether they are preceded by other text in the comment. The compiler only recognizes the keywords when they are the first identifier following an optional initial series of blanks and/or horizontal tabs. In addition, `lint` recognizes only a single digit of the `VARARGS` count. The compiler accumulates as many digits as appear in the count.

Types and Sizes

[Table 3.14](#) shows information for the supported data types, including:

- The size for each data type in memory
- The size for each data type in the two register types (the Dn 40-bit data register and the Rn 32-bit address register)
- The required alignment for each data type
- The value range for each data type.

Table 3.14 Data types and sizes

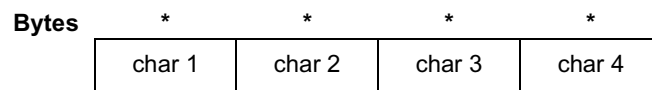
Type	Size (in Bits)				Range	
	Memory	Dn	Rn	Align	Minimum	Maximum
char	8	40	32	8	-128	127
unsigned char	8	40	32	8	0	255
short	16	40	32	16	-32,768	32,767
unsigned short	16	40	32	16	0	65,535
int	32	40	32	32	-2,147,483,648	2,147,483,647
unsigned int	32	40	32	32	0	4,294,967,295
long	32	40	32	32	-2,147,483,648	2,147,483,647
unsigned long	32	40	32	32	0	4,294,967,295
float, double, and long double	32	40	32	32	-1.17E-38	1.17E+38
fractional short	16	40	-	16	-1	0.99969842
fractional long / int	32	40	-	32	-1	0.9999999953
pointer	32	40	32	32	0	0xFFFFFFFF

NOTE Fractional short and fractional long/int are not language types. These types can be used with intrinsic functions only and map to the predefined types short and long/int, respectively.

Characters

A character, whether signed or unsigned, is stored in memory in one byte (8 bits), and is always aligned on an 8-bit boundary. Arrays of characters occupy one byte per character. [Figure 3.5](#) shows the memory layout for characters.

Figure 3.5 Characters—memory layout



When loaded into registers, signed characters are signed extended, while unsigned characters are zero extended. [Figure 3.6](#) illustrates

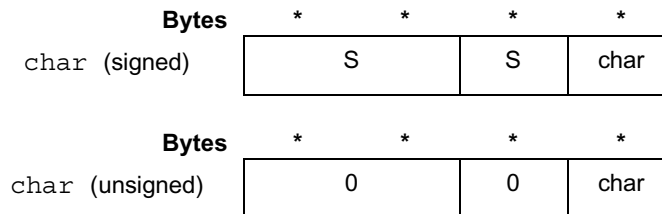
the layout for signed and unsigned characters in the Dn (40-bit) data register. “S” indicates the signed extension of the value.

Figure 3.6 Characters—Dn register layout



[Figure 3.7](#) shows the layout for signed and unsigned characters in the Rn (32-bit) address register.

Figure 3.7 Characters—Rn register layout



Integers

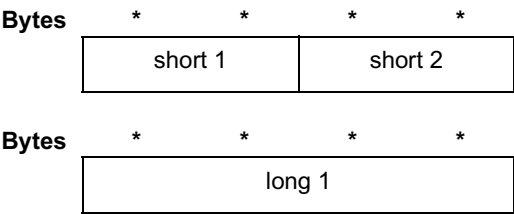
Integer arithmetic is performed using data sizes appropriate to the arithmetic operation. Short integers use at least 16-bit wide operations (single-precision integer arithmetic), and long integers use at least 32-bit (double-precision integer arithmetic).

Short and long integers are stored in memory using little-endian representation (the least significant bits in the lower address), unless the option `-be` is specified.

Integer arithmetic overflow wraps around and does not result in any additional side effects.

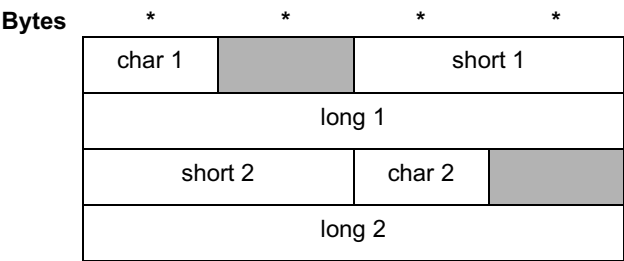
[Table 3.15](#) shows the memory layout for short and long integers.

Table 3.15 Integers—memory layout



Short integers must be aligned on 2-byte (16-bit) boundaries, while long integers must be aligned on a 4-byte (32-bit) boundary. [Figure 3.8](#) illustrates the alignment of short and long integers, in conjunction with characters.

Figure 3.8 Integers—alignment



As with characters, when loaded into registers, signed integers are signed extended, while unsigned integers are zero extended.

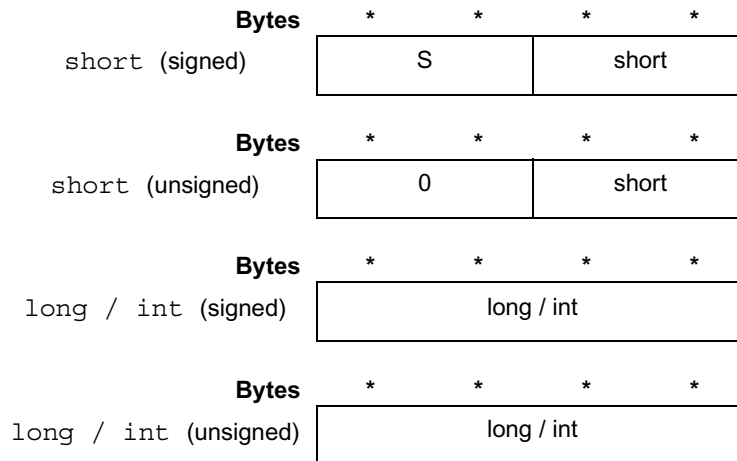
[Figure 3.9](#) illustrates the layout for signed and unsigned short and long integers in the Dn (40-bit) data register. “S” indicates the signed extension of the value.

Figure 3.9 Integers—Dn register layout



[Figure 3.10](#) shows the layout for signed and unsigned short and long integers in the Rn (32-bit) address register.

Figure 3.10 Integers—Rn register layout



Floating point

Floating point, double, and long double type integers are mapped to a single precision IEEE-754 type, using 32 bits (4 bytes). The compiler generates calls for library functions to evaluate floating point expressions. The representation of these integers in memory and in the registers is exactly the same as for long integers.

Fractional representation

Since C does not provide built-in support for fractional types, the syntactic representation of fractional types and operations is implemented by intrinsic functions using integer data types.

Fixed -point arithmetic is performed using 16-bit, 32-bit, 40-bit, and 64-bit operations. Fractional integers are stored in memory using little-endian representation, meaning the least significant bits in the lower address, unless the option `-be` is specified.

Fractional type overflows may saturate and do not result in any additional side effect. Rounding and saturation modes are determined as part of the startup code, or with optional intrinsic function calls.

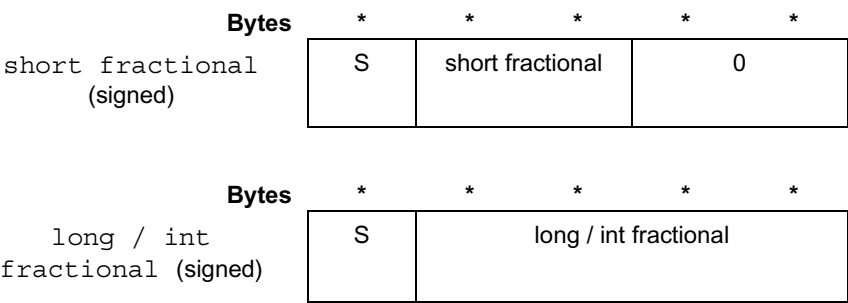
Operations on double and extended precision type objects are limited to assignments and fractional arithmetic using intrinsic

functions only. Integer operations on extended precision types are not supported.

Fractional types are mapped to their corresponding predefined types. A fractional short maps to the predefined type short, a fractional long maps to the predefined type long, and a fractional int maps to the predefined type int.

[Figure 3.11](#) illustrates the layout for fractional short and long integers in the Dn (40-bit) data register, which is the only register used for fractional integer types. “S” indicates the signed extension of the value.

Figure 3.11 Fractional integers—Dn register layout



When loading data from memory into data registers, the compiler aligns the data in the registers according to the context in which the data is used.

Pointers

Pointers contain addresses of data objects or functions. Pointers are represented in memory using 32 bits (4 bytes). In the small memory model, although pointers are represented in memory using 32 bits, only 16 bits are meaningful. The representation of pointers in memory and in the registers is exactly the same as for unsigned long integers.

Bit-fields

Members of structures are always allocated on byte boundaries, and are aligned according to their fundamental base type. However, bit-fields in structures can be allocated at any bit and of any length not exceeding the size of a long word (32 bits). Signed and unsigned bit-fields are permitted and are sign extended when fetched. A bit-field of type int is considered signed.

Bit-fields are always allocated from the low-address end of a word (right to left or little-endian), even if the option `-be` is specified. Bit-field sizes are not allowed to cross a long word boundary.

In [Listing 3.19](#), the structure `more` has 4-byte alignment and will have a size of 4 bytes. This is because the bit-fields in the structure are governed by the fundamental type `long` which requires a 4-byte alignment.

Listing 3.19 Bit-field alignment to long word (1)

```
struct more {  
    long first : 3;  
    unsigned int second : 8;  
};
```

The structure `less` shown in [Listing 3.20](#) requires only a one byte alignment because this is the requirement of the fundamental type `char` used in this structure.

Listing 3.20 Bit-field alignment to character

```
struct less {  
    unsigned char third : 3;  
    unsigned char fourth : 8;  
};
```

The alignments are driven by the underlying type, not the width of the fields. These alignments are to be considered along with any other structure members.

In [Listing 3.21](#), the structure `careful` requires a 4-byte alignment; its bit-fields require only a one byte alignment, but the field `fluffy` requires a 4-byte alignment because its fundamental type is `long`.

Listing 3.21 Bit-field alignment to long word (2)

```
struct careful {  
    unsigned char third : 3;  
    unsigned char fourth : 8;  
    long fluffy;  
};
```

Fields within structures and unions begin on the next possible suitably aligned boundary for their data type. For fields that are not bit-fields, this is a suitable byte alignment. Bit-fields begin at the next available bit offset, with the following exception: the first bit-field after a member that is not a bit-field will be allocated on the next available byte boundary.

In [Listing 3.22](#), the offset of the field `c` is one byte. The structure itself has 4-byte alignment and is four bytes in size because of the alignment restrictions introduced by using the `long` underlying data type for the bit-field.

Listing 3.22 Bit-field offset

```
struct s {  
    int bf: 5;  
    char c;  
}
```

Fractional and Integer Arithmetic

The ability to perform both integer and fractional arithmetic is one of the strengths of the Metrowerks Enterprise C compiler.

Fractional arithmetic is typically required for computation-intensive algorithms such as digital filters, speech coders, vector and array processing, digital control, or other signal processing tasks. In this mode, the data is interpreted as fractional values, and the computations are performed interpreting the data as fractional. Fractional arithmetic examples are shown in [Figure 3.12](#).

Figure 3.12 Fractional arithmetic examples

```
0.5 * 0.25    -> 0.125  
0.625 + 0.25  -> 0.875  
0.125 / 0.5   -> 0.25  
0.5 >> 1      -> 0.25
```

Often, saturation is used when performing calculations in this mode to prevent the severe distortion that occurs in an output signal generated from a result where a computation overflows without saturation. Saturation can be selectively enabled or disabled so that

intermediate calculations can be performed without limiting, and limiting is only done on final results.

NOTE The notation used in [Figure 3.12](#) is for illustration purposes only because C does not support the specification of fractional constants using floating-point notation. The compiler implements fractional arithmetic using intrinsic functions based on integer data types.

Integer arithmetic is invaluable for controller code, array indexing and address computations, peripheral setup and handling, bit manipulation, and other general purpose tasks, as shown in [Figure 3.13](#).

Figure 3.13 Integer arithmetic examples

```
4 * 3      -> 12
1201 + 79  -> 1280
63 / 9     -> 7
100 << 1   -> 200
```

Data in a memory location or register can be interpreted as fractional or integer, depending on the needs of a user's program. [Table 3.16](#) shows how a 16-bit value can be interpreted as either a fractional or integer value, depending on the location of the binary point.

NOTE The binary representation shown in [Table 3.16](#) corresponds to the location of the binary point when interpreting the data as fractional. If the data is interpreted as integer, the binary point is located immediately to the right of the LSB.

Table 3.16 Interpretation of 16-bit data values

Binary Representation	Hexadecimal Representation	Integer Value (Decimal)	Fractional Value (Decimal)
0.100 0000 0000 0000	0x4000	16384	0.5
0.010 0000 0000 0000	0x2000	8192	0.25
0.001 0000 0000 0000	0x1000	4096	0.125
0.111 0000 0000 0000	0x7000	28672	0.875

Table 3.16 Interpretation of 16-bit data values

Binary Representation	Hexadecimal Representation	Integer Value (Decimal)	Fractional Value (Decimal)
0.000 0000 0000 0000	0x0000	0	0.0
1.100 0000 0000 0000	0xC000	-16384	-0.5
1.110 0000 0000 0000	0xE000	-8192	-0.25
1.111 0000 0000 0000	0xF000	-4096	-0.125
1.001 0000 0000 0000	0x9000	-28672	-0.875

The following equation shows the relationship between a 16-bit integer and a fractional value:

$$\text{Fractional Value} = \text{Integer Value} / (2^{15})$$

There is a similar equation relating 40-bit integers and fractional values:

$$\text{Fractional Value} = \text{Integer Value} / (2^{31})$$

[Table 3.17](#) shows how a 40-bit value can be interpreted as either an integer or fractional value, depending on the location of the binary point.

Table 3.17 Interpretation of 40-bit data values

Hexadecimal Representation	40-Bit Integer in Entire Accumulator	16-Bit Integer in MSP (Decimal)	Fractional Value (Decimal)
0x0 4000 0000	1073741824	16384	0.5
0x0 2000 0000	536870912	8192	0.25
0x0 0000 0000	0	0	0.0
0xF C000 0000	-1073741824	-16384	-0.5
0xF E000 0000	-536870912	-8192	-0.25

The following code fragment illustrates the use of integer arithmetic:

Listing 3.23 Integer arithmetic computation

```
a = a + b*c;
```

[Listing 3.24](#) provides an example of the use of an intrinsic function to implement fractional arithmetic.

Listing 3.24 Fractional arithmetic computation

```
a = L_mac(a,b,c);
```

Intrinsic Functions

The compiler supports a large number of intrinsic (built-in) functions that map directly to SC100 assembly instructions. As C does not support fractional types and operations, these intrinsic functions enable fractional operations to be implemented using integer data types.

The syntax of the compiler group of intrinsic functions is structured for full compatibility with the ETSI and ITU reference implementations of bit-exact standards.

Data types for intrinsic functions

The following four data types are defined for specific use with intrinsic functions:

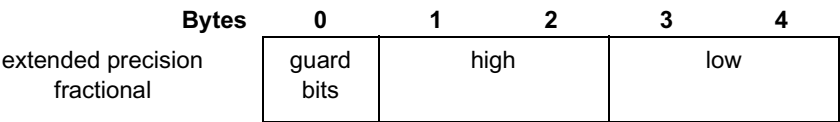
- Fractional short, a 16-bit fractional value mapped to a short
- Fractional long, a 32-bit fractional value mapped to a long
- Extended precision fractional, a 40-bit value which can be used only in intrinsic functions
- Double precision fractional, a 64-bit value which can be used only in intrinsic functions

Extended and double precision fractional types enable algorithms to be defined which require precision larger than 32 bits. These data types can be used only with intrinsic functions and with assignments. Variables defined as extended and double precision fractionals cannot be used for standard arithmetical or other operations.

Extended precision fractional

The extended precision fractional (`Word40`) is a 40-bit data type which occupies the entire Dn (40-bit) register, as shown in [Figure 3.14](#):

Figure 3.14 Extended precision fractional—Dn register layout



This data type is mapped in the compiler as a structure containing two elements:

- A 32-bit integer placed to the right of the binary point.
- An 8-bit integer placed to the left of the binary point. These “guard bits” can be used to ensure a more accurate result when an overflow occurs.

When stored in memory, an extended precision fractional variable occupies 64 bits. The least significant 32 bits are stored in the first 32-bit word, and the 8 most significant guard bits are stored in the second 32-bit word in an undefined position.

Double precision fractional

The double precision fractional data type (`Word64`) consists of 64 bits, all of which are assumed to be to the right of the binary point. This data type is mapped in the compiler as a structure containing two 32-bit elements.

Fractional constants

Fractional constants require integer notation, since floating point notation is not supported. For example, to express the value 0.5 as a fractional constant, the integer representation in hexadecimal must be used in the source code, in this case `0x4000`.

Initializing variables with fractional values

Variables can be initialized as fractional values, using the following macros:

- `WORD16` initializes a value as a fractional short.
- `WORD32` initializes a value as a fractional long.

For example, `short x = WORD16(0.5)` initializes `x` as a fractional short with the value `0x4000`.

Intrinsic function categories

The following categories of intrinsic functions are provided:

- Fractional arithmetic
- Long fractional arithmetic
- Double precision fractional arithmetic
- Extended precision fractional arithmetic, with guard bits
- Architecture primitives
- Architecture primitives that generate identical assembly instructions
- Bit reverse addressing

[Table 3.18](#) lists and describes each group of intrinsic functions.

Table 3.18 Intrinsic functions

Intrinsic Function	Declaration	Description
Fractional arithmetic		
add	short add(short, short)	Short add
sub	short sub(short, short)	Short subtract
mult	short mult(short, short)	Short multiply
div_s	short div_s(short, short)	Short divide
mult_r	short mult_r(short, short)	Multiply and round
L_mac	long L_mac(long, short, short)	Multiply accumulate
mac_r	short mac_r(long, short, short)	Multiply accumulate and round
L_msu	long L_msu(long, short, short)	Multiply subtract
msu_r	short msu_r(long, short, short)	Multiply subtract and round
abs_s	short abs_s(short)	Short absolute value
negate	short negate(short)	Short negate
round	short round(long)	Round
shl	short shl(short, short)	Short shift left
shr	short shr(short, short)	Short shift right
shr_r	short shr_r(short, short)	Short shift right and round
norm_s	short norm_s(short)	Normalize any fractional value
max	short max(short, short)	Maximum value of any two short fractional values

Using the Metrowerks Enterprise C Compiler

Intrinsic Functions

Table 3.18 Intrinsic functions (*continued*)

Intrinsic Function	Declaration	Description
min	short min(short, short)	Minimum value of any two short fractional values
saturate	short saturate(short)	Short saturation
Long fractional arithmetic		
L_add	long L_add(long, long)	Long add
L_sub	long L_sub(long, long)	Long subtract
L_mult	long L_mult(short, short)	Long multiply
extract_h	short extract_h(long)	Extract 16 MSB of long word
extract_l	short extract_l(long)	Extract 16 LSB of long word
L_deposit_h	long L_deposit_h(short)	Deposit short in MSB
L_deposit_l	long L_deposit_l(short)	Deposit short in LSB
L_abs	long L_abs(long)	Long absolute value
L_negate	long L_negate(long)	Long negate
norm_l	short norm_l(long)	Normalize any long fractional value
L_max	long L_max(long, long)	Maximum value of any two long fractional values
L_min	long L_min(long, long)	Minimum value of any two long fractional values
L_shl	long L_shl(long, short)	Long shift left
L_shr	long L_shr(long, short)	Long shift right
L_shr_r	long L_shr_r(long, short)	Long shift right and round
L_sat	long L_sat(long)	Long saturation
Double precision fractional arithmetic		
D_mult	Word64 D_mult(long, long)	Double precision multiply
D_mac	Word64 D_mac(Word64, long, long)	Double precision multiply accumulate
D_msu	Word64 D_msu(Word64, long, long)	Double precision multiply subtract
D_add	Word64 D_add(Word64, Word64)	Double precision add
D_sub	Word64 D_sub(Word64, Word64)	Double precision subtract
D_cmpeq	short D_cmpeq(Word64, Word64)	Double precision compare equal
D_cmpgt	short D_cmpgt(Word64, Word64)	Double precision compare greater than

Table 3.18 Intrinsic functions (*continued*)

Intrinsic Function	Declaration	Description
D_sat	Word64 D_sat (Word64)	Double precision saturation
D_round	long D_round (Word64)	Double precision round
D_set	Word64 D_set (long, unsigned long)	Concatenate two longs into one double precision value
D_extract_l	unsigned long D_extract_l (Word64)	Extract 32 LSB of double precision value
D_extract_h	long D_extract_h (Word64)	Extract 32 MSB of double precision value
Extended precision fractional arithmetic (with guard bits)		
X_mult	Word40 X_mult (short, short)	Short multiply to long long word
X_mac	Word40 X_mac (Word40, short, short)	Short multiply accumulate to long long word
X_msu	Word40 X_msu (Word40, short, short)	Short multiply subtract to long long word
X_set	Word40 X_set (char, unsigned long)	Concatenate char and unsigned long into one long long word
X_add	Word40 X_add (Word40, Word40)	Long add including guard bits
X_sub	Word40 X_sub (Word40, Word40)	Long subtract including guard bits
X_shl	Word40 X_shl (Word40, short)	Long shift left with guard bits
X_shr	Word40 X_shr (Word40, short)	Long shift right with guard bits
X_extract_h	short X_extract_h (Word40)	Extract 16 MSB of long long word
X_extract_l	short X_extract_l (Word40)	Extract 16 LSB of long long word
X_round	short X_round (Word40)	Round long long value
X_norm	short X_norm (Word40)	Normalize any long long fractional value
X_rol	Word40 X_rol (Word40)	Rotate left a long long word
X_ror	Word40 X_ror (Word40)	Rotate right a long long word
X_abs	Word40 X_abs (Word40)	Long absolute value with guard bits
X_sat	long X_sat (Word40)	Long saturation including guard bits
X_or	Word40 X_or (Word40, Word40)	Logical OR two long values with guard bits
X_trunc	long X_trunc (Word40)	Truncate guard bits

Using the Metrowerks Enterprise C Compiler

Intrinsic Functions

Table 3.18 Intrinsic functions (*continued*)

Intrinsic Function	Declaration	Description
X_extend	Word40 X_extend(long)	Sign extend long value to include guard bits
X_cmpeq	short X_cmpeq(Word40, Word40)	Fractional compare equal with guard bits
X_cmpgt	short X_cmpgt(Word40, Word40)	Fractional compare greater than with guard bits
Architecture primitives		
L_rol	long L_rol(long)	Rotate left a long
L_ror	long L_ror(long)	Rotate right a long
mpyuu	long mpyuu(long, long)	Long multiply 16 LSB of two long words, treating both words as unsigned values
mpyus	long mpyus(long, long)	Long multiply 16 LSB of the first long word, treated as an unsigned value, by 16 MSB of the second long word, treated as signed
mpysu	long mpysu(long, long)	Long multiply 16 MSB of the first long word, treated as a signed value, by 16 LSB of the second long word, treated as unsigned
setnosat	setnosat()	Set saturation mode off
setsat32	setsat32()	Set saturation mode on
set2crm	set2crm()	Set rounding mode to two's-complement rounding mode
setcnvrm	setcnvrm()	Set rounding mode to convergent rounding mode
Architecture primitives that generate identical assembly instructions		
debug	void debug()	Enter Debug mode
debugev	void debugev()	Generate Debug event
mark	void mark()	If trace buffer enabled, write program counter to trace buffer
stop	void stop()	Enter Stop low power mode
trap	void trap()	Execute Trap exception
wait	void wait()	Enter Wait low power mode
ei	void ei()	Enable interrupts

Table 3.18 Intrinsic functions (*continued*)

Intrinsic Function	Declaration	Description
di	void di()	Disable interrupts
illegal	void illegal()	Execute illegal exception
Bit reverse addressing		
InitBitReverse	InitBitReverse	Allocate a bit reverse iterator
BitReverseUpdate	BitReverseUpdate	Increment the iterator with bit reverse
EndBitReverse	EndBitReverse	Free bit reverse iterator

Intrinsic functions examples

[Listing 3.25](#) illustrates the use of a number of intrinsic functions.

Listing 3.25 Intrinsic functions

```
#include <prototype.h>
void Iir(short Input[], short Coef[], short FiltOut[])
{
    long L_Sum;
    short int Stage, Smp;

    FiltOut[0] = Input[0];
    for (Smp = 1; Smp < S_LEN; Smp++)
    {
        L_Sum = L_msu(LPC_ROUND, FiltOut[Smp - 1], Coef[0]);

        for (Stage = 1; ((0 < (Smp - Stage)) && Stage < NP); Stage++)
            L_Sum = L_msu(L_Sum, FiltOut[Smp - Stage - 1], Coef[Stage]);

        L_Sum = L_shl(L_Sum, ASHIFT);
        L_Sum = L_msu(L_Sum, Input[Smp], 0x8000);
        FiltOut[Smp] = extract_h(L_Sum);
    }
}
```

[Listing 3.26](#) illustrates the use of extended precision variables and intrinsic functions using guard bits.

Listing 3.26 Intrinsic functions using extended precision

```
#include <prototype.h>
docorr()
{
    int j, i;
    int shift_val;
    short corr_0;

    Word40 E_acc, E_sum;

    E_acc = X_extend(0);
    E_sum = X_extend(0);

    for (i = 0; i < M1; i++)
    {
        for (j = 0; j < M2; j++)
            E_acc = X_mac (E_acc, sample[j], coeff[j] );

        L_sample[i] = X_sat(E_acc);
        E_acc = X_abs(E_acc);
        E_sum = X_add(E_sum, E_acc);
    }

    shift_val = X_norm(E_sum);
    corr_0 = 0;

    for (i = 0; i < M1; i++)
    {
        sample[i] = round (L_shr (L_sample[i], shift_val));
        corr_0 = sub (corr_0, sample[i]);
    }

    corr = corr_0;
}
```

Pragmas

Pragmas allow you greater control over your application, enabling you to give the compiler specific additional information about how to process certain statements. The pragmas that you specify in your code provide the compiler with context-specific hints which can save the compiler unnecessary operations, and help to further enhance the optimization process.

You can include as many pragmas as necessary in your source code. The sections that follow describe the syntax and placement rules for pragmas.

Syntax

The pragmas supported by the compiler have the following general syntax:

```
#pragma pragma-name [argument(s)]
```

One or more of the arguments may be optional. Arguments are comma-delimited.

Each pragma must fit on one line.

Placement

Each pragma applies only in a certain context and you must place each one accordingly. Several categories of pragmas exist:

- Pragmas that apply to functions can appear only in the scope of the function, after the opening “{”.
- Pragmas that apply to statements must be placed immediately before the relevant statement, or immediately before any comment lines which precede the statement.
- Pragmas that apply to variables must follow the object definition, or any comment lines which follow that definition. Objects referred to by pragmas must be explicitly defined.
- In addition, other pragmas exist that do not fit into the preceding categories.

The pragmas supported by the compiler are listed in [Table 3.19](#).

Using the Metrowerks Enterprise C Compiler

Pragmas

Table 3.19 Pragmas

Pragma	Description
Function Pragmas	
<code>#pragma inline</code>	Forces function inlining.
<code>#pragma noinline</code>	Disables function inlining.
<code>#pragma save_ctxt</code>	Forces save and restore of all registers that are used in this procedure.
<code>#pragma external func</code> [name = <i>string</i> , convention = <i>number</i> , nosideeffects]	Defines a function as external to the C application, or as a function that can be called from outside the application.
<code>#pragma interrupt func</code>	Defines the specified function as an interrupt handler.
<code>#pragma safe_mod</code>	Forces the compiler to perform modulo optimization without checking whether the initial value belongs to the modulo range.
<code>#pragma inline</code>	Causes the compiler to always inline the function in which this pragma appears.
<code>#pragma noinline</code>	Causes the compiler to never inline the function in which this pragma appears.
<code>#pragma dynamic</code>	Always maps the function in which it appears on a dynamic stack regardless of optimizations.
Pragmas That Apply to Statements	
<code>#pragma profile value</code>	Sets profiling information for a statement.
<code>#pragma loop_count (lower_bound, upper_bound, {2/4}, remainder)</code>	Specifies the minimum and maximum limits for a loop, the loop count divider (2 or 4), and the use of the remainder.
Pragmas That Apply to Variables	
<code>#pragma align var_name {4/8}</code>	Forces stricter alignment on an object. Needed for paired moves.
<code>#pragma align *var_name {4/8}</code>	Indicates that the address of the variable referenced by a pointer is aligned as specified.
Other Pragmas	
<code>#pragma opt_level "optimization_level"</code>	Controls the level of code optimization. Can apply at either a function level or a module level. Valid values for the optimization level are O0, O1, O2, O3, O3s.

Table 3.19 Pragmas (*continued*)

Pragma	Description
<code>#pragma pgm_seg_name "name"</code>	Rename the text segment in the ELF file. (You must define the name used to override the default in the linker command file.)
<code>#pragma data_seg_name "name"</code>	Rename the data segment in the ELF file. (You must define the name used to override the default in the linker command file.)
<code>#pragma rom_seg_name "name"</code>	Rename the rom segment in the ELF file. (You must define the name used to override the default in the linker command file.)
<code>#pragma bss_seg_name "name"</code>	Rename the bss segment in the ELF file. (You must define the name used to override the default in the linker command file.)
<code>#pragma init_seg_name "name"</code>	Rename the init segment in the ELF file. (You must define the name used to override the default in the linker command file.)
<code>#pragma call_conv call_conv_name func_name</code>	<p>Specify the calling convention for the compiler to use on a given function. The application configuration file must define the calling convention name, and <i>func_name</i> must be a previously declared function.</p> <p>An example follows:</p> <pre>#pragma call_conv My_Call_Conv func1</pre>
<code>#pragma default_call_conv call_conv_name</code>	Specify the calling convention for a module (file). The application configuration file must define the calling convention name.
<code>#pragma align var_name value</code>	<p>Aligns the named variable on constant byte boundaries. Examples follow:</p> <ul style="list-style-type: none"> • <code>#pragma align MyVar 8</code> • <code>#pragma align *MyVar 8</code>

Table 3.19 Pragmas (*continued*)

Pragma	Description
<code>#pragma external func_name [no_side_effect]</code>	Indicates that the named function is external. The optional argument <code>no_side_effect</code> tells the compiler that this function has no side effects. Examples follow: <ul style="list-style-type: none"> • <code>#pragma external __send</code> • <code>#pragma external printf no_side_effect</code>
<code>#pragma interrupt func_name</code>	Specifies the named function as an interrupt entry. An example follows: <code>#pragma interrrupt func1</code>
<code>#pragma loop_count (min_val, max_val)</code>	Specifies the minimum and maximum number of iterations for the loop in which this pragma appears. An example follows: <code>#pragma loop_count (10, 100)</code>
<code>#pragma loop_unroll constant_val</code>	Unrolls constant time in the loop in which this pragma appears. Applies only to a single instruction block loop without calls. An example follows: <code>#pragma loop_unroll 2</code>
<code>#pragma loop_unroll_and_jam constant_val</code>	Causes the compiler to perform an unroll and jam on the enclosed loop nest. The constant is the unroll factor. This #pragma applies only to loop nests that are single instruction blocks without calls. An example follows: <code>#pragma loop_unroll_and_jam 8</code>

Pragmas that apply to functions

The pragmas in this category provide additional information about specific functions, and are defined in the scope of the function to which they apply, directly after the “{” which marks the start of the scope.

Forcing or disabling function inlining

Inlining enables the compiler to improve optimization by replacing a function call by the entire function. For very small functions, for

example, where the overhead of the function call is greater than the size of the function itself, this can be very efficient.

You can use `#pragma inline` to force the compiler to inline a specific function, or `#pragma noinline` to prevent the compiler from inlining a certain function. In the code segment shown in [Listing 3.27](#), any calls to the function which follows `#pragma noinline` will not be inlined.

Listing 3.27 `#pragma noinline`

```
static int proc_30(int a)
{

#pragma noinline

    int tab_30[1000];

    tab_30[0] = 4*a;
    return(tab_30[0]);
}
```

Saving the entire context of the system

During normal processing, the compiler saves the contents of registers that have been changed, and any other essential data. You can force the compiler to save the entire context of the machine, including all registers that are used in this procedure, so that it can be restored if necessary to its previous state, at the exact point at which the specific function started to execute.

Using `#pragma save_ctxt` to save the entire system status can incur a large overhead, and should only be used where absolutely necessary.

[Listing 3.28](#) illustrates the use of `#pragma save_ctxt` to force the compiler to save the complete machine context upon entry to the specified function.

Listing 3.28 `#pragma save_ctxt`

```
void EntryPoint()
{
#pragma save_ctxt
```

```
...  
}
```

Defining a function as external

When the compiler encounters an unresolved function call, it assumes by default that this is a call to an external function that exists outside the application. The pragma `#pragma external` enables you to:

- Confirm this assumption, by informing the compiler that the call is to an external function defined outside the application
- Define the function as an internal function that can be called from outside the application

The effect of the pragma depends on its placement, as described below:

- If `#pragma external` is specified in the global scope, the compiler does not expect to find the body of the function within the current application. The compiler uses standard calling conventions to call the function, and does not issue warnings for unresolved references. Specifying `#pragma external` in the global scope is valid only with cross-file optimization.
- If `#pragma external` is specified within the function scope, followed by the body of the defined function, the compiler recognizes this as an internal function that can be called from outside the application.

The following optional parameters can be specified with `#pragma external`:

- Specify `name = string` to provide a specific function name, to override the default linkage name allocated to the function.
- Define `convention = number` to select the calling convention to be used instead of the default standard convention.
- Specify `nosideeffects` if the function does not change any variable values in the application, and can be moved or duplicated in other parts of the application without making any changes.

When `nosideeffects` is specified, the compiler does not need to make worst case assumptions about any possible impact that the function may have within the application.

In the first part of [Listing 3.29](#), `printf` is defined as an external function that does not exist within the application, and that has no effect on any variables in the application. In the second part of the example, the function `ICanBeCalled` is defined inside the application and may be called by external function calls. This function therefore has to obey the standard calling conventions.

Listing 3.29 `#pragma external`

```
extern void printf();
#pragma external printf [nosideeffects]

void main()
{
    printf("Hello there\n");
}

void ICanBeCalled(int X, int Y)
{
#pragma external ICanBeCalled [name ="xyz"]
    ...
}
```

Defining a function as an interrupt handler

A function that operates as an interrupt handler differs from other functions in three basic respects:

- It must save and restore all resources that it uses, as it can be called at any time an interrupt occurs, and cannot assume any conventions.
- It runs in “exception” mode, which forces the compiler to generate instructions that are slightly different from the instructions issued in normal mode.
- It cannot be passed parameters nor return a value.

You can use `#pragma interrupt` to define a function as an interrupt handler, as shown in [Listing 3.30](#).

Listing 3.30 `#pragma interrupt`

```
void IntHandler();
#pragma interrupt IntHandler
extern long Counter;
```

```
void IntHandler()
{
Counter++;
}
```

Force modulo optimization

Previously, the compiler performed modulo optimization only if the following conditions were true:

- The initial value belonged to the modulo range.
- The step was static and smaller than $2 * modulo_value$.

Now, you can use the `safe_mod` pragma to force the compiler to skip checking the initial value before performing modulo optimization. Place the `safe_mod` pragma in the function.

The syntax for the `safe_mod` pragma follows:

```
#pragma safe_mod
```

NOTE The `safe_mod` pragma applies to all modulo candidates in the affected function.

[Listing 3.31](#) shows a code example that uses the `safe_mod` pragma.

Listing 3.31 `#pragma safe_mod`

```
int func1(unsigned int init, short *pt, short *pt1, int Max)
{
#pragma safe_mod
    unsigned int  i, j;
    int  Acc = 0;

    i = init;
    for(j = 0; j<Max; j++, i++) {
        Acc = pt[i%3] + pt1[i%5];
    }

    for(j = 0; j<Max; j++, i++) {
        Acc = pt[i%3] + pt1[i%5];
    }
    return Acc;
}
```

}

Without the pragma, the compiler could not perform modulo transformation because `init` is unknown (for the first loop) and because `i` in the second loop comes from the first loop and is also unknown.

NOTE You still must pass the `-mod` option and enable optimizations because modulo replacement relies heavily on loop analysis (which is not done at optimization level `-O0`).

Pragmas that apply to statements

Pragmas which apply to statements are placed immediately before the relevant statement.

Specifying a profile value

By default, the profiler provided with the compiler enables it to make the necessary assumptions about the number of times to execute a given statement. You can specify `#pragma profile`, followed by a value and immediately preceding a statement, to specify to the compiler the exact number of times that the statement executes.

In [Listing 3.32](#), the value following `#pragma profile` notifies the compiler that the loop executes only 10 times. If `#pragma profile` is not specified, the compiler assumes that, since this is a loop with dynamic bounds, the loop executes 25 times (the default). It is important to note that this assumption affects the optimization of the program, and not its correctness.

Listing 3.32 `#pragma profile` with constant value

```
#include <prototype.h>
int energy (short block[], int N)
{
    int i;
    long int L_tmp = 0;

    for (i = 0; i < N; i++)
#pragma profile 10
```

```
    L_tmp = L_mac (L_tmp, block[i], block[i]);

return round (L_tmp);
}
```

With if-then-else constructs, `#pragma profile` can be used to inform the compiler which branch executes more frequently, and the frequency ratio between the two branches, meaning the number of times one branch executes in relation to the other.

In [Listing 3.33](#), the two `#pragma profile` statements have the values 5 and 50. These values notify the compiler that the `else` branch section executes 10 times more frequently than the first (implied `then`) section. When used in this way, the exact `#pragma profile` values are not significant, since they indicate the frequency ratio, and not the absolute values. In this example, the values 1 and 10 would convey the same information.

Listing 3.33 `#pragma profile` with frequency ratio

```
#include <prototype.h>
int energy (short block[], int N)
{
    int i;
    long int L_tmp = 0;

    if ( N>50)
#pragma profile 5
        for (i = 0; i < 50; i++)
            L_tmp = L_mac (L_tmp, block[i], block[i]);
    else
#pragma profile 50
        for (i = 0; i < N; i++)
            L_tmp = L_mac (L_tmp, block[i], block[i]);

    return round (L_tmp);
}
```

Defining a loop count

The compiler tries to evaluate the number of times a loop iterates using the static information available. In cases where this static

information is not supplied to the compiler, if you know the upper and lower limits of a loop, you can use `#pragma loop_count` to provide these values. Supplying such information, which cannot always be discerned automatically by the compiler, enables generation of more efficient code.

Similarly, specifying a divider for the loop count enables the optimizer to unroll loops in the most efficient way. The loop count can be divided by either 2 or 4, corresponding to the number of execution units. You can also instruct the compiler whether to use the remainder, if there is one following division of the loop count, to execute the loop an additional number of times.

[Listing 3.34](#) shows the syntax of `#pragma loop_count`.

Listing 3.34 Syntax of `#pragma loop_count`

```
#pragma loop_count (lower_bound, upper_bound,  
                  [{2/4}, [remainder]])
```

Define a value for `lower_bound` for the minimum number of times the loop will iterate, and a value for `upper_bound` for the maximum number of times.

The divider parameter is optional. Only the values 2 or 4 may be specified as the divider.

To specify that a remainder should be used for the loop count, specify a value for `remainder`. The `remainder` argument is only valid if a value has been specified for the divider.

The pragma `#pragma loop_count` must be placed inside the loop to which it relates, and outside any nested loops which the loop contains.

In [Listing 3.35](#), the loop will always iterate at least 4 times and at most 512 times. The iteration count will always be divisible by 4. As no remainder is specified, any remainder from the division will be disregarded.

Listing 3.35 `#pragma loop count`

```
void correlation2 (short vec1[], short vec2[],  
                  int N, short *result)  
{
```

```
long int L_tmp = 0;
int i;

for (i = 0; i < N; i++)
    #pragma loop_count (4,512,4)
    L_tmp = L_mac (L_tmp, vec1[i], vec2[i]);

*result = round (L_tmp);
}
```

Pragmas that apply to variables

These pragmas are placed immediately after the definition of the object(s) to which they refer. Objects referred to by pragmas must first be explicitly defined.

Alignment of variables

Objects are usually aligned according to their size. The default alignment for arrays is determined by their base type.

An array may need to be aligned to a specified value before it can be passed to an external function. The pragma `#pragma align` can be used to force the alignment of arrays passed to an external function, to meet the specific alignment requirements of the function.

To force the alignment of an array before passing it to an external function, specify `#pragma align`, followed by the defined array object, and either the value 4 for 4-byte (32-bit double word) alignment or 8 for 8-byte (64-bit quad word) alignment.

Certain instructions, such as `move .2w` and `move .4w`, which move words in pairs, may require alignment to be applied that is stricter than the alignment defined for the data types involved.

In certain cases, the compiler cannot assess the alignment for dynamic objects and has to assume that the objects have the alignment requirements for their base type. As a result, the compiler cannot use the multiword move instructions for these objects. By specifying the exact alignment for one or more objects, you can enable the compiler to use these multiword moves and generate more efficient code.

You can use the pragma `#pragma align` to provide the compiler with specific alignment information about pointers to arrays, in order to enable the compiler to use multiword move instructions.

To inform the compiler that the address of an array is aligned as required for multiword moves, specify `#pragma align`, followed by the pointer to the array object, and either the value 4 for 4-byte alignment or 8 for 8-byte alignment. When using `#pragma align` in this way, you should ensure that the object is in fact aligned as required, since this form of the pragma does not force the alignment.

In the first part of [Listing 3.36](#), array `a` is forced to 8-byte alignment before being passed to the external function `Energy`. The second part of the example informs the compiler that both input vectors are aligned to 32 bits. The instruction `move .2f` may be used here.

Listing 3.36 `#pragma align`

```
#include <prototype.h>
short a[10];
#pragma align a 8

extern int Energy( short a[] );
int foo()
{
    return Energy(a);
}

short Cor(short vec1[], short vec2[], int N)
{
    #pragma align *vec1 4
    #pragma align *vec2 4

    long int L_tmp = 0;
    long int L_tmp2 = 0;
    int i;

    for (i = 0; i < N; i += 2)
        L_tmp = L_mac(L_tmp, vec1[i], vec2[i]);

    L_tmp2 = L_mac(L_tmp2, vec1[i+1], vec2[i+1]);
    return round(L_tmp + L_tmp2);
}
```

Other Pragmas

This section discusses additional pragmas that are available.

Optimization level control by means of pragma in the source code

Optimization Level

The `opt_level` pragma can apply to a single function or to the whole module. To apply `opt_level` to a function, place the pragma in the function body. To apply `opt_level` to a module, place the pragma at the module level.

An `opt_level` pragma in a function supercedes an `opt_level` pragma at the module level. An `opt_level` pragma at the module level supercedes the optimization level passed by the shell.

[Listing 3.37](#) shows the possible `opt_level` pragma statements.

Listing 3.37 Possible `opt_level` pragma statements

```
# The following statement is equivalent to scc -O0.  
#pragma opt_level = "O0"  
# The following statement is equivalent to scc -O1.  
#pragma opt_level = "O1"  
# The following statement is equivalent to scc -O2.  
#pragma opt_level = "O2"  
# The following statement is equivalent to scc -O3.  
#pragma opt_level = "O3"  
# The following statement is equivalent to scc -Os -O3.  
#pragma opt_level = "O3s"
```

You cannot use `-O3` as a command-level option with the `O0`, `O1`, `O2`, and `Os` options. You can use `-O3` only with `O3s`.

The `O0`, `O1`, `O2`, and `Os` options used at command-level are compatible with `O0 O1 O2 Os` as pragmas.

[Listing 3.38](#) shows a code example that uses the `opt_level` pragma.

For [Listing 3.38](#), if the command-line is `scc -Os opt.c`, the compiler compiles `func1` in `O0` as the module-level option is `O0`. The compiler compiles the `func2` function in `O2` (which overrides `O0` specified in the module and `Os` specified in the command line).

Listing 3.38 opt.c: opt_level pragma code example

```
typedef struct {
    int a;
    int b;
} S;

#pragma opt_level = "O0"

void func1()
{
    typedef struct {
        short a;
        short b;
    } S;

    S v;

    v.a = 0;
    v.b = 1;
}

void func2()
{
#pragma opt_level = "O2"
    S v;

    v.a = 2;
    v.b = 3;
}
```

Renaming text segment in the ELF file

To rename the text segment in the ELF file, use the `pgm_seg_name` pragma. The `pgm_seg_name` pragma has the following syntax:

```
#pragma pgm_seg_name "name"
```

NOTE The new segment name that you define cannot include any spaces.

You can place the `pgm_seg_name` pragma anywhere in the module (file), and it affects the entire file.

You must define the name used to override the default segment name in the linker command file.

Renaming data segment in the ELF file

To rename the data segment in the ELF file, use the `data_seg_name` pragma. The `data_seg_name` pragma has the following syntax:

```
#pragma data_seg_name "name"
```

NOTE The new segment name that you define cannot include any spaces.

You can place the `pgm_seg_name` pragma anywhere in the module (file), and it affects the entire file.

You must define the name used to override the default segment name in the linker command file.

Renaming rom segment in the ELF file

To rename the rom segment in the ELF file, use the `rom_seg_name` pragma. The `rom_seg_name` pragma has the following syntax:

```
#pragma rom_seg_name "name"
```

NOTE The new segment name that you define cannot include any spaces.

You can place the `rom_seg_name` pragma anywhere in the module (file), and it affects the entire file.

You must define the name used to override the default segment name in the linker command file.

Renaming bss segment in the ELF file

To rename the bss segment in the ELF file, use the `bss_seg_name` pragma. The `bss_seg_name` pragma has the following syntax:

```
#pragma bss_seg_name "name"
```

NOTE The new segment name that you define cannot include any spaces.

You can place the `bss_seg_name` pragma anywhere in the module (file), and it affects the entire file.

You must define the name used to override the default segment name in the linker command file.

Renaming init segment in the ELF file

To rename the init segment in the ELF file, use the `init_seg_name` pragma. The `init_seg_name` pragma has the following syntax:

```
#pragma init_seg_name "name"
```

NOTE The new segment name that you define cannot include any spaces.

You can place the `init_seg_name` pragma anywhere in the module (file), and it affects the entire file.

You must define the name used to override the default segment name in the linker command file.

Predefined Macros

The compiler shell maintains a number of predefined macros, including standard C macros, and additional macros which are specific to the Metrowerks Enterprise C compiler and the SC100 architecture. [Table 3.20](#) lists these predefined macros.

Table 3.20 Predefined macros

Macro Name	Description
<code>__LINE__</code>	The line number of the current source line.
<code>__FILE__</code>	The name of the current source file.
<code>__DATE__</code>	The compilation date, as a character string in the form <code>Mmm dd yyyy</code> (for example, Jan 23 1999).
<code>__TIME__</code>	The compilation time, as a character string in the form: <code>hh:mm:ss.t</code>
<code>__STDC__</code>	Decimal constant 1, indicating ANSI conformance.
<code>__STDC_VERSION__</code>	Defined in ANSI C mode as 199409L.
<code>__SIGNED_CHARS__</code>	Defined when char is signed by default
<code>__VERSION__</code>	The version number of the compiler, as a character string in the form <code>nn.nn</code> .
<code>__INCLUDE_LEVEL__</code>	Decimal constant, indicating the current depth of file inclusion.

Using the Metrowerks Enterprise C Compiler

Predefined Macros

Table 3.20 Predefined macros (*continued*)

Macro Name	Description
<code>_ENTERPRISE_C_</code>	<p>Defined for use with the Enterprise compiler. If your source file may be compiled with other compilers apart from the Enterprise, this macro should be included in a conditional statement to ensure that the appropriate commands are activated, for example:</p> <pre>#ifdef _ENTERPRISE_C_ (Enterprise-specific commands) #else #endif</pre>
<code>_SC100_</code>	<p>Defined for use with all compilers based on the SC100 architecture. If your source file may be compiled with other compilers apart from those based on the SC100 architecture, this macro should be included in a conditional statement to ensure that the appropriate commands are activated, as shown in the following example:</p> <pre>#ifdef _SC100_ (SC100-specific commands) #else #endif</pre>
<code>_SC110_</code> <code>_SC140_</code>	<p>The architecture variant, which specifies the number of MAC units to be used by the compiler:</p> <ul style="list-style-type: none">• <code>_SC110_</code> indicates 1 MAC unit.• <code>_SC140_</code> indicates 4 MAC units. <p>Only one of these macros is valid for each invocation of the compiler. The macro that is selected, and the value of the architecture variant, are determined by the value set for the <code>-arch</code> option when the compiler is invoked. If no value is specified for <code>-arch</code>, the default is SC140 (<code>_SC140_</code>).</p>

Interfacing C and Assembly Code

The Metrowerks Enterprise C compiler supports interfacing between C source code and assembly code, enabling access to functionality not provided by C. This chapter describes the features of this interface and provides instructions, guidelines, and examples.

This chapter contains the following topics:

- [Inlining a Single Assembly Instruction](#)
- [Inlining a Sequence of Assembly Instructions](#)
- [Calling an Assembly Function in a Separate File](#)
- [Including Offset Labels in the Output File](#)

Inlining a Single Assembly Instruction

A single assembly instruction can be inlined in a sequence of C statements and compiled by the compiler. To ensure successful compilation of an inlined assembly instruction, note the following guidelines:

- The compiler passes an inlined instruction to the assembly output file in the form of text, and therefore has no knowledge of the contents or side effects of the instruction. It is important that you ensure that there is no risk of the instruction affecting the C and/or assembly environment and producing unpredictable results. For example, you should not use an inlined assembly instruction to change the contents of registers, as the compiler has no knowledge of such changes. Similarly, you should not include any jumps or labels which access the C code and may affect the correctness of the tracking algorithms.
- Inlined assembly code instructions are ignored by the optimizer.

- Since the compiler treats the assembly instruction as a string of text, it cannot perform any error checking on the instruction. Check the syntax and text of the instruction carefully prior to compilation. Errors in assembly code are identified only at the assembly stage of the compilation process.
- A single inlined assembly instruction cannot reference a C object. The only way to reference a C object in assembly code is by inlining a sequence of assembly instructions.

To inline a single assembly instruction, use the `asm` statement. The syntax is as for a standard function call, with one argument enclosed in double quotation marks, as shown in [Listing 4.1](#).

Listing 4.1 Inlining a single assembly instruction

```
asm("wait");
```

Inlining a Sequence of Assembly Instructions

It is possible to use assembly code that references C objects, by defining a separate function that consists of a sequence of assembly instructions, and inlining this in your C code. Such a function is implemented entirely in assembly and may not include C statements, but can accept parameters referenced by the assembly code.

Guidelines for Inlining Assembly Code Sequences

The following guidelines are similar to those for the inlining of individual assembly instructions and apply also to the use of inlined sequences of assembly code:

- The compiler passes a sequence of inlined instructions to the assembly output file as a string of text, and therefore has no knowledge of the contents or side effects of the instructions. It is important that you ensure that the assembly function does not affect the C and/or assembly environment and does not produce unpredictable results. For example, do not use inlined assembly instructions to change the contents of registers, and do not alter the sequence of C code instructions by specifying jumps, as the compiler has no knowledge of such changes.

- Functions based on inlined sequences of assembly code cannot be used by the optimizer, and are ignored during optimization. Avoid using assembly-based functions if a C alternative is available, in order to ensure maximum optimization of the code.
- The compiler performs no error checking on the sequence of assembly instructions. Assembly code errors are identified only at the assembly stage of the compilation process.

The guidelines listed below apply specifically to the use of inlined sequences of assembly code:

- When passing parameters to an inlined sequence of assembly instructions, registers are not automatically allocated. You must specify for each parameter the register in which the parameter enters or exits the function. There is no need to save and restore the registers before and after the function.
- The compiler is unable to deduce whether an inlined function is likely to affect the application, for example, if it modifies global variables. It is important that you provide the compiler with this information if there is a possibility that the function may have any side effects.
- A function that is initially defined as stand-alone may in certain circumstances be included in another sequence of instructions. Inlined functions should therefore not use statements such as RTS. If the function is used in a sequence of instructions, the compiler adds the necessary return statements automatically.
- Local variables are not automatically allocated by the compiler for use by assembly functions. If the function requires the use of local variables, you must allocate these specifically on the stack or define them as static variables.
- Assembly functions defined as a sequence of instructions can access global variables in the C source code, since these are static by definition.

Defining an Inlined Sequence of Assembly Instructions

When defining a sequence of inlined assembly instructions, you define the header for the function before the body of the instructions, and you specify the registers to be used by each parameter. You can define a list of read parameters, a list of write parameters, and/or a list of modified registers, as appropriate.

Interfacing C and Assembly Code

Defining an Inlined Sequence of Assembly Instructions

[Listing 4.2](#) shows the syntax for inlining a sequence of assembly instructions.

Listing 4.2 Syntax for inlining a sequence of assembly instructions

```
asm <func prototype>
{
asm_header
    optional arg binding
    optional return value
    optional read list
    optional write list
    optional modified reg list
asm_body
    <asm code>
asm_end
}

optional arg binding
    .arg
        <ident> in <reg>;
        <ident> in <reg>;
        ...
optional return value
    return in <reg>

optional read list:
    .read <ident>,<ident>,...;

optional write list:
    .write <ident>,<ident>,...;

optional modified reg list:
    .reg <reg>, <reg>, ...;
```

The following syntax conventions apply:

- Identifiers must have the prefix `_` (underscore).
- Registers must have the prefix `$` (dollar sign).
- Labels must have the suffix `.` (period).

[Listing 4.3](#) shows the syntax for an inlined assembly function that takes two arguments as input parameters and returns one value. The first argument is passed in the register d0, and the second parameter in the register r1. The result is returned in d0.

Listing 4.3 Inlining syntax

```
asm int t6( int param1, int *param2)
{
asm_header
.arg
    _param1 in $d0;
    _param2 in $r1;
return in $d0;
.reg $d0,$d1,$r1;
asm_body
    move.l (r1),d1
    add    d0,d1,d0
asm_end
}
```

In [Listing 4.4](#), the function t6 accepts two parameters, an integer p1 passed in register d14, and a pointer p2 passed in r7. The result of the function is returned in d14.

Listing 4.4 Simple inlined assembly function

```
#include <stdio.h>
int A[10] = {1,2,3,4,5,6,7,8,9,0};
asm int t6(int p1, int *p2)
{
asm_header
.arg
    _p1 in $d14;
    _p2 in $r7;
return in $d14;
.reg $d14,$d1,$r7;
asm_body
    move.l (r7),d1
    add    d14,d1,d14
asm_end
}
```

Interfacing C and Assembly Code

Defining an Inlined Sequence of Assembly Instructions

```
int main()
{
    int k = 8;
    int s;

    s = t6(k,&A[3]);

    printf("S= %d\n",s);

return s;
}
```

[Listing 4.5](#) shows the use of labels and hardware loops within inlined assembly functions. You should use hardware loops within assembly functions only if you know that the loop nesting is legal. In this example, the function is called from outside a loop, and the use of hardware loops is therefore allowed.

Listing 4.5 Inlined assembly function with labels and hardware loops

```
#include <stdio.h>
char sample[10] = {9,6,7,1,0,5,1,8,2,6};
int status;

asm char t7(int p)
{
    asm_header
    .arg
    _p in $d7;
return in $d8;
    .reg $d7,$d8,$r1;
asm_body

    clr      d8                move.l    #_sample,r1                doen3
d7
    dosetup3 _L10
    loopstart3
_L10:
    move.b    (r1),d1
    add      d8,d1,d8
    inc      d1
    move.b    d1,(r1)+
```

```
    loopend3

asm_end
}

int main()
{
    int m = 8;
    int s,i;

    for(i=0;i < 10;i++) {
        sample[i] *= 2;
        printf("%d ",sample[i]);
    }
    printf("\n");
    s = (int)t7(m);
    printf("S= %d\n",s);

    for(i=0;i < 10;i++)
        printf("%d ",sample[i]);
    printf("\n");
    return 1;
}
```

[Listing 4.6](#) shows how global variables are referenced within an inlined assembly function. Global variables are accessed using their linkage name, which is by default the variable name prefixed by the character `_` (underscore). The variables `vector1` and `vector2` are therefore accessed within the function as `_vector1` and `_vector2` respectively.

Listing 4.6 Referencing global variables in an inlined assembly function

```
#include <stdio.h>

short vector1[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int vector2[] = {11,12,13,14,15,16,17,18,19,1,2,3,4,5,6};
short result_1=0;
int result_2=0;

asm void test(int n, short *r1,int *r2)
{
```

Interfacing C and Assembly Code

Calling an Assembly Function in a Separate File

```
asm_header
.arg
    _n   in $r1;
    _r1  in $r3;
    _r2  in $r7;
.reg $d0,$r1,$r6,$r11,$r3,$r7;
asm_body
    move.l    #_vector1,r6
    move.l    #_vector2,r11
    addl1a    r1,r6
    addl2a    r1,r11
    move.w    (r6),d0
    asrr      #<2,d0
    move.w    d0,(r3)
    move.l    (r11),d1
    asl       d1,d2
    move.l    d2,(r7)
asm_end
}

int main(void)
{
    test(12,&result_1,&result_2);

    printf("Status = %d %d\n", (int)result_1, result_2);

    return (int)result_2;
}
```

Calling an Assembly Function in a Separate File

The compiler supports calls to assembly functions that are contained in separate files, and enables you to integrate these files with your C application.

To include a call to an assembly function in your program, follow the steps described below:

- 1 Write the assembly function in a separate file from your C source files. Use the standard calling conventions.

- 2 If required, assemble the file. This step is optional.
- 3 In your C source file, define the assembly function as an external function.
- 4 Specify both the C source file and the assembly file as input files in the shell command line to integrate the files during compilation.

The following examples show how a segment of C code calls a function that performs an FFT algorithm implemented in assembly.

Writing the Assembly Code

[Listing 4.7](#) shows the assembly code for the FFT algorithm, in the file `fft.sl`.

Listing 4.7 Assembly function in a separate file

```
;  
; extern void fft(short *, short*);  
;  
; Parameters:  pointer to input buffer in r0  
;              pointer to output buffer in r1  
;  
_fft:  
; Save and restore d6, d7, r6, r7, according to  
; calling conventions.  
push        d6  
push        d7  
push        r6  
push        r7  
  
< implementation of FFT algorithm >  
  
pop         r6  
pop         r7  
pop         d6  
pop         d7  
  
rts
```

Calling the Assembly Function

The C code that calls the FFT function is shown in [Listing 4.8](#). This source code is saved in the file `test_fft.c`.

Listing 4.8 C code calling assembly function

```
#include <stdio.h>
extern void fft(short *, short*);
#pragma external fft

short in_block[512];
short out_block[512];

int in_block_length, out_block_length;

void main()
{
    int i;
    FILE *fp;
    int status;

    in_block_length=512;
    out_block_length=512;

    fp=fopen("in.dat","rb");
    if( fp== 0 )
    {
        printf("Can't open parameter file: input_file.dat\n");
        exit(-1);
    }

    printf("Processing function fft \n");

    while ((status=fread(in_block, sizeof(short), in_block_length,
fp)) == in_block_length)
    {
        fft(in_block,out_block);
    }
}
```

Integrating the C and Assembly Files

[Listing 4.9](#) shows how the two input files are specified in the shell command line:

Listing 4.9 Integrating C and assembly files

```
scc -o test_fft.eld test_fft.c fft.sl
```

Including Offset Labels in the Output File

In some cases when assembly functions are called, data structures need to be shared between the C source code and the assembly code. In [Listing 4.10](#), the layout of the structure `complex` needs to be used by the assembly code.

Listing 4.10 Data structure shared between C and assembly

```
struct complex
{
    short r;
    short i;
};

struct complex CVEC1, CVEC2;
volatile struct complex res;

void main()
{
    cmpy (&CVEC1, &CVEC2, &res);
}
```

The `-do` option in the shell command line instructs the compiler to include the details of C data structures in the output assembly file. You can specify this as an additional option in the command line, as shown in [Listing 4.11](#):

Listing 4.11 Specifying the output of offset information

```
scc -o test.eld test.c cmpy.sl -do
```

Interfacing C and Assembly Code

Including Offset Labels in the Output File

When the `-do` option is specified, the output file shows the offsets for all field definitions in each data structure defined in the C source code. The symbolic label is composed of:

`<module name>_<structure name>_<field name>`, as shown in the following example:

Listing 4.12 Data structure offsets in the assembly output file

<code>test_complex_r</code>	<code>equ</code>	<code>0</code>
<code>test_complex_i</code>	<code>equ</code>	<code>2</code>

The symbolic labels in the output file can be used in the assembly code, making the code more readable, as shown in [Listing 4.13](#).

Using these symbolic labels also makes maintenance of the assembly code easier when changes are made to the C code.

Listing 4.13 Using symbolic offsets in assembly code

```
=====
; Function cmpy
;
; Parameter x      passed in r0
; Parameter y      passed in r1
; Parameter result passed in (sp-12)
;=====

        global      _cmpy
        align       2

_cmpy
[
    move.2f    (r0),d0d1
    move.2f    (r1),d2d3
]
[
    mpy        d0,d2,d5
    mpy        d0,d3,d7
    macr       -d1,d3,d5
    macr       d1,d2,d7
    move.l     sp-12),r2
]
    rtsd
```

```
moves.f d5, (r2+test_complex_r)
moves.f d7, (r2+test_complex_i)
```

Optimization Techniques and Hints

This chapter explains how the SC100 optimizer operates, and describes the optimization levels and individual optimizations which can be applied.

This chapter contains the following topics:

- [Optimizer Overview](#)
- [Using the Optimizer](#)
- [Optimization Types and Functions](#)
- [Guidelines for Using the Optimizer](#)
- [Optimizer Assumptions](#)

Optimizer Overview

The SC100 optimizer converts preprocessed source files into assembly output code, applying a range of code transformations which can significantly improve the efficiency of the executable program. The goal of the optimizer is to produce output code which is functionally equivalent to the original source code, while improving its performance in terms of execution time and/or code size.

Code Transformations

The optimizer is extremely versatile, and can transform the code in a number of ways to achieve optimal results. These code transformations include:

- Substituting instructions with more efficient code
- Removing redundant instructions

- Inserting instructions to simplify operations
- Hoisting or lowering instructions to reduce unnecessary operations

[Table 5.1](#) illustrates each of these instruction transformations.

Table 5.1 Instruction transformation

Transformation Type	Before Optimization	After Optimization
Substitution	<code>move.w #0,d0</code>	<code>clr d0</code>
Removal	<code>move.w #0,d0</code> <code>move.w (r1),d0</code>	<code>move.w (r1),d0</code>
Insertion and removal	<code>for(i=0; i<3; i++)</code> <code>foo();</code>	<code>foo();</code> <code>foo();</code> <code>foo();</code>
Hoisting	<code>if(a<3)</code> <code>Tbit = TRUE;</code> <code>else</code> <code>Tbit = FALSE;</code>	<code>Tbit = FALSE;</code> <code>if(a<3)</code> <code>Tbit = TRUE;</code>

Basic Blocks

The majority of the code transformations operate on basic blocks of code. A basic block of code is a linear sequence of instructions for which there is only one entry point and one exit point. There are no branches in a basic block. In general, bigger basic blocks enable better optimization, since the scope for further optimization is increased.

Linear and Parallelized Code

The optimizer can produce code that takes full advantage of the multiple execution units provided by the SC100 architecture.

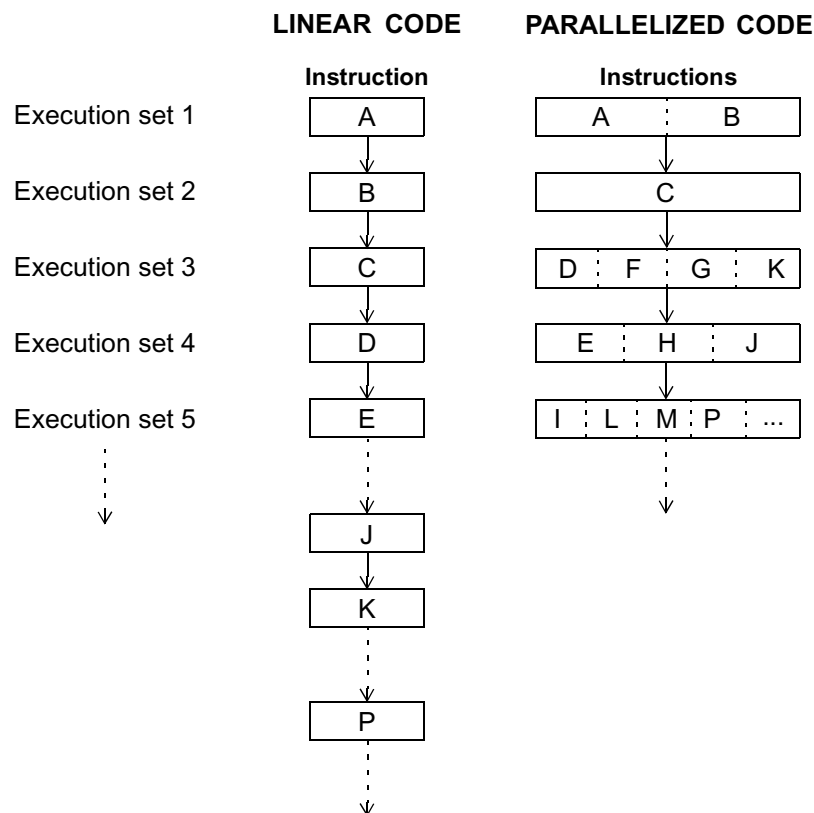
Executable programs process instructions in the form of execution sets, with one execution set per cycle. The optimizer can increase the number of instructions in an execution set, enabling two or more execution units to process instructions in parallel, in the same cycle. In this way, linear code is transformed into parallelized code:

- **Linear code** uses only one execution unit, regardless of the number of units available. Each execution set consists of one instruction only.

- **Parallelized code** execution sets can comprise multiple instructions which execute in parallel using the available number of execution units. Parallelized code executes faster and more efficiently than linear code.

[Figure 5.1](#) illustrates the transformation of linear code, comprising a series of single instruction execution sets, into parallelized code, which consists of execution sets containing one or more instructions each:

Figure 5.1 Linear and parallelized code



Dependencies between instructions can restrict the level of parallelization that the optimizer can achieve.

Optimization Levels and Options

Three basic optimization levels are provided, all of which maintain a balance between code density and speed:

- Level 0 compiles the fastest and produces the slowest output as linear code. Level 0 produces unoptimized code.
- Level 1 takes longer to compile, applies target-independent optimizations, and produces optimized linear code.
- Level 2 (the default) compiles more slowly than Level 1, applies all target-independent optimizations, as well as all target-specific optimizations, and can produce faster, parallelized code.

Only one of the above optimization options may be selected for each compilation.

Two supplemental optimizations are available which may be used in combination with Level 1 or Level 2 optimization:

- Space optimization enables you to apply the indicated level of optimization, while weighting the optimization process in favor of program size. Programs or modules that have been optimized for space require a smaller amount of memory but may sacrifice program speed.
- Cross-file optimization is a complex process which requires significantly more compilation time than non-cross file optimization. With cross-file optimization, the optimizer applies the required level of optimization across all the files in the application at the same time, and as a result produces the most efficient program code.

Cross-file optimization is generally applied at the end of the development cycle, after all source files have been compiled and optimized individually or in groups. By default, the optimizer operates without cross-file optimization.

[Table 5.2](#) summarizes the optimization options.

Table 5.2 Optimization options summary

Option	Description	Benefits
-O0 (Level 0)	<ul style="list-style-type: none"> Disables all optimizations. Outputs non-optimized, linear assembly code. 	<ul style="list-style-type: none"> Compiles fastest.
-O1 (Level 1)	<ul style="list-style-type: none"> Performs all target-independent (non-parallelized) optimizations, such as function inlining. Omits all target-specific optimization steps. Outputs optimized, linear code. 	<ul style="list-style-type: none"> Compiles faster than option -O2 (the default). Produces faster programs than option -O0. Generates assembly code which correlates clearly with the C source code, and can assist debugging.
-O2 (Level 2) (Default)	<ul style="list-style-type: none"> Performs all optimizations. Outputs optimized, non-linear assembly code. 	<ul style="list-style-type: none"> Takes advantage of parallel execution units, producing the highest performance code possible without cross-file optimization.
-O3	<ul style="list-style-type: none"> Performs the same optimizations as the -O2 option and global register allocation. (This option causes virtual register allocation to be used rather than physical register allocation.) 	<ul style="list-style-type: none"> The number of cycles is decreased.
-Os	<ul style="list-style-type: none"> Performs the indicated level of optimization, with emphasis on reducing code size. Can be specified together with any of the other optimization options except -O0. 	<ul style="list-style-type: none"> Produces optimized assembly code which is small.
-Og	<ul style="list-style-type: none"> Performs cross-file optimization. Can be specified together with any of the other optimization options except -O0. Produces the most efficient results when specified with the -O2 (default) option. Compiles significantly slower than the other options. 	

Using the Optimizer

By default, the compiler optimizes all source code files using Level 2 optimization without cross-file optimization. You can choose to optimize your source code at the level that you require at each stage of program development, and you can optimize individual sections of the program according to their purpose in the application. For example, you may wish to prepare your application as follows:

- **During initial development stages:** Use the default Level 2 optimization to compile your source code files, individually or in groups. If required, optimize certain sections of the application for maximum speed, and optimize other sections for size, to reduce the memory space they occupy.
- **During final development stages:** Select Level 2 and cross-file optimization, in order to apply all optimizations across the entire application. The compilation is slower, but produces the most effective optimization results.

You select the optimization level and mode to be applied by specifying one or more options in the shell command line.

Invoking the Optimizer

The optimizer can be invoked by including the required options in the shell command line or command file, as illustrated in the examples that follow.

The command line shown in [Listing 5.1](#) invokes the optimizer with one input source file, and the default optimization settings. The optimizer applies Level 2 optimizations without cross-file optimization, with a balance between space optimizations and speed.

Listing 5.1 Invoking the optimizer with default settings

```
scc -o file.eld file.c
```

[Listing 5.2](#) shows how to invoke the optimizer with the Level 1 option, to apply target-independent optimizations only. The optimizer maintains a balance between space optimizations and speed, and operates without cross-file optimization.

Listing 5.2 Invoking the optimizer for target-independent optimizations only

```
scc -O1 -o file.eld file.c
```

The command line shown in [Listing 5.3](#) invokes the optimizer in cross-file optimization mode. The optimizer processes all the specified source files together, applying the default Level 2 optimizations to all the modules in the application.

Listing 5.3 Invoking the optimizer with cross-file optimization

```
scc -Og -o file.eld file1.c file2.c file3.c
```

Optimizing for Space

Your application, or specific parts of it, may require code that occupies the least possible space in memory. You can optimize the file(s) for space at the expense of program speed.

To activate space optimization, specify the `-Os` option in the shell command line.

The `-Os` option generates the smallest code size for the given optimization level. If no optimization level is specified with `-Os`, the `-O2` optimization level is selected by default.

All optimizations associated with the current optimization level are applied, except those that adversely affect code size. (In fact, optimizations that reduce code size are emphasized.)

Depending on your application, the best code density might be achieved using other optimization combinations, such as `-O2` and `-Og`.

Using Cross-File Optimization

Once you have optimized your individual source files and groups of files, you can invoke the optimizer in cross-file mode to ensure maximum optimization across the entire application, in order to produce the most efficient code.

With cross-file optimization, all the code in the application is processed by the compiler at the same time. The optimizer has no need to make worst case assumptions since all the necessary

information is available. This enables the optimizer to achieve an extremely powerful level of optimization.

The main disadvantages of compiling with cross-file optimization are the high consumption of resources required, and the slow compilation time. In addition, because of the interdependency that cross-file optimization creates between all segments of the application, the entire application needs to be recompiled if any one source code file is changed. For these reasons, cross-file optimization is generally used at the final stage of development.

To activate cross-file optimization, specify the `-Og` option in the shell command line. While you can specify this option with any of the other optimization-level options, cross-file optimization is generally recommended with optimization Level 2. The `-O2` option is the default and may be omitted.

Optimization Types and Functions

The optimizer implements two main types of optimization:

- Target-independent optimizations improve the output code without taking into account the properties of the target machine.
- Target-specific optimizations achieve code improvements by exploiting the architecture features of the target machine.

Both sets of optimizations can be applied to individual files and groups of files, with or without cross-file optimization.

Changes in the code as a result of one optimization may enable another optimization to be applied, producing an accumulative effect.

Dependencies and Parallelization

Dependency between instructions directly limits how successfully the optimizer can apply the various optimizations. An instruction is considered to be dependent on another if a change in their order of execution influences the result of the operation.

The optimizer can group instructions into parallelized execution sets only if these instructions do not contain dependencies. Parallelization of different parts of the program, or of iterations of

the same loop, can significantly increase the speed of the executable application.

[Listing 5.4](#) illustrates a simple dependency between two instructions. The value of d0 is entirely different when the order of these instructions is reversed. These instructions cannot be executed in parallel.

Listing 5.4 Simple instruction dependency

```
move.w  #5,d0      ; Sets register d0 to 5
add     d0,d1,d2    ; Adds the values in d0 and d1 into register d2
```

An example of dependency arising from an algorithm is shown in [Listing 5.5](#). The value of the variable sum must be calculated before it can be used in the L_mac instruction.

Listing 5.5 Algorithm instruction dependency

```
sum = mpy(a,b) ;
result = L_mac(sum,c,d) ;
```

The optimizer can operate most effectively with code which contains as few dependencies as possible.

Target-Independent Optimizations

In the high-level optimization phase, a number of general, target-independent optimizations are implemented. All target-independent optimizations are applied when either optimization Level 1 (option -O1) or the default optimization Level 2 (option -O2) is selected.

These target-independent optimizations are summarized in [Table 5.3](#), and examples of each are given in the sections that follow.

For a detailed discussion of the principles behind target-independent optimizations, refer to *Compilers Principles, Techniques, and Tools*, by Aho, Sethi, and Ullman.

Table 5.3 Summary of target-independent optimizations

Optimization	Description
Strength reduction (loop transformations)	Transforms array access patterns and induction variables in loops, and replaces them with pointer accesses
Function inlining	Substitutes a function call with the code of the function
Common subexpression elimination	Replaces an expression with its value if it occurs more than once
Loop invariant code	Moves code outside a loop if its value is unchanged by the loop
Constant folding and propagation	Calculates the value of an expression at compilation time if it contains known static constants
Jump-to-jump elimination	Combines jump instructions
Dead code elimination	Removes code that is never executed
Dead storage/assignment elimination	Removes redundant variables and value assignments

The output from the target-independent optimizations is in the form of linear assembly code.

Strength reduction (loop transformations)

The purpose of strength reduction is to increase the effectiveness of the code by transforming operations which are “expensive” in terms of resources, into less expensive, linear operations. For example, addition and subtraction are linear functions which require less operation cycles than multiplication and division.

When an address calculation that contains multiplication is replaced by one containing addition, the amount of resources required by the code is significantly reduced, since addition can be implemented using the complex addressing mode of the Address Generation Unit (AGU). Where the multiplication appears within a loop, the benefit of the replacement is further increased.

The strength reduction optimization identifies and transforms induction variables, meaning variables whose successive values form an arithmetic progression, usually within a loop. An example

of an induction variable is a subscript which points to the addresses of array elements, and increases with each iteration of the loop. The computation of such a variable can be moved to a position outside the loop to avoid repeated operations, and/or transformed for use with linear operations.

Simple and complex loops and array access patterns are transformed where possible into simpler, linear forms, as described in the sections that follow.

Simple loops

[Figure 5.2](#) shows the generated pseudocode and output assembly code for a simple loop which initializes an array. The loop structure is static, meaning that its induction variables, the loop counter `i` and the array offset `t1`, both increase by increments of known constant values.

Figure 5.2 Loop transformation - simple loop

C source code

```
int table[100];
step = 1;

for(i=0; i<100; i+=step)
    table[i] = 0;
```

Pseudocode before optimization

```

    i = 0;
L1   t1 = i * 4;
     table[t1] = 0;
     i++;
     if(i<100) goto L1
```

Pseudocode after optimization

```

    i = 0;
     t1 = i * 4;
L1   table[t1] = 0;
     t1 = t1 + 4;
     i++;
     if(i<100) goto L1
```

Assembly code output

```
move.l #_table,r0    clr d2
loopstart3
    move.l d2,(r0)+
loopend3
```

Before optimization, the calculation of the value of `t1` is within the loop, and is incremented by multiplication. After optimization, the initial value of `t1` is set outside the loop, and its value is incremented inside the loop by addition. The resulting values are identical for both forms, but in the optimized version the resource overhead is considerably lower.

The same principles also apply to more complex loop structures and array access patterns, as described in the sections that follow:

- Dynamic loops, in which increments are based on a variable whose value is not known at compilation time
- Multi-step loops, in which the loop iterator increments more than once in each iteration of the loop
- Composed variable loops, in which one or more variables or iterators are linked to each other in a linear relationship
- Square loops, which access elements in a two-dimensional array as in a matrix, on a row-by-row basis
- Triangular loops, which are similar to square loops, but which access each row in the matrix from an incremented starting position in each subsequent row

Dynamic loops

In a dynamic loop, one or more increments are based on variables whose values are not known at compilation time.

[Figure 5.3](#) shows the generated code for a dynamic loop in which the value of the loop increment and its upper limit are not known at the time of compilation. The optimization removes the initial multiplication instruction from the body of the loop, and inside the loop the multiplication increment instruction is replaced by an addition instruction.

Figure 5.3 Loop transformation - dynamic loop

C source code

```
step = step_table[1];
for(i=0; i<MAX; i+=step)
    table[i] = 0;
```

Pseudocode before optimization

```
step = step_table[1];
i = 0;
L1    t1 = i * 2;
      table[t1] = 0;
      i = i + step;
      if(i<MAX) goto L1
```

Pseudocode after optimization

```
i = 0;
step = step_table[1];
t1 = i * 2;
t2 = step * 2;
L1    table[t1] = 0;
      t1 = t1 + t2;
      i = i + step;
      if(i<MAX) goto L1
```

Assembly code output

```
L2      clr     d3          add     d1,d0,d1
        move.l d3,(r1)     cmpge.w #100,d1
        adda   r2,r1
        jf     L2
```

Multi-step loops

Loops in which the loop iterator increments more than once in each iteration of the loop are defined as multi-step loops.

In the multi-step loop shown in [Figure 5.4](#), the loop iterator *i* increments twice within the loop. In this case, *i* is transformed into an induction variable which increments in linear progression in three stages.

Figure 5.4 Loop transformation - multi-step loop

C source code

```
int table[10];
for(i=0; i<10; i++)
    table[i] = i;
    i++;
    table[i] = 0;
```

Pseudocode before optimization

```
L1      i = 0;
        t1 = i * 2;
        table[t1] = i;
        i = i + 1;
        t2 = i * 2;
        table[t2] = i;
        i = i + 1;
        if(i<10) goto L1
```

Pseudocode after optimization

```
i = 0;
t1 = i * 2;
t2 = i * 2 + 2;
t3 = i;
Repeat 10 times:
    table[t1] = t3;
    table[t2] = 0;
    t1 = t1 + 4;
    t2 = t2 + 4;
    t3 = t3 + 2;
```

Assembly code output

```
        loopstart3
L93
        move.l    d0,(r0)+n3      add     #<2,d0

        move.l    d2,(r1)+n3
        loopend3
```

Composed variable loops

A composed variable loop incorporates one or more variables or iterators which have a linear relationship between them. The loop transformation optimizes such loops by moving the multiplication instruction to a position outside the loop, and by substituting one of the variables with a constant.

This optimization can be applied only when the variables are linked by linear arithmetic functions, meaning those calculations involving addition or subtraction of the variables, or multiplication of a variable by a constant. Functions which include non-linear operations, such as multiplication of two induction variables, cannot be optimized in this way.

[Figure 5.5](#) illustrates the generated code for a composed variables loop. In this example the increment is the result of a linear calculation using the two induction variables *i* and *j*.

Figure 5.5 Loop transformation - composed variables

C source code

```
int table[100];
for(i=0, j=0; i<10; i++)
    table[10 * i + j] = i;
    j++;
```

Pseudocode before optimization

```
i = 0;
j = 0;
t1 = i * 10;
L1 t2 = t1 + j;
   t3 = t2 * 2; /* address */
   table[t3] = i;
   i = i + 1;
   t1 = t1 + 10;
   j = j + 1;
   if(j < 10) goto L1
```

Pseudocode after optimization

```
i = 0; j = 0;
t1 = 1 * 10;
t2 = t1 + j;
t3 = t2 * 2;
Repeat 10 times:
    table[t3] = i;
    i = i + 1;
    t3 = t3 + 22;
```

Assembly code output

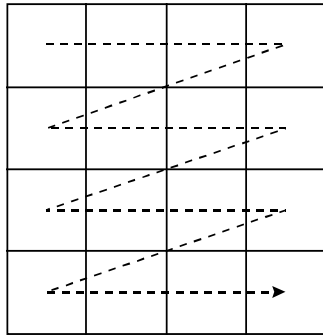
```
loopstart3
L93
move.l    d0, (r0)+n3    inc    d0
loopend3
```

Square loops

A square loop is a multi-dimensional array access pattern which is similar to a matrix in which cells are accessed horizontally in rows. The access can start at the first cell in each row, or the optimizer can process block access, in which the access may not begin at the first cell and may not end at the last cell.

The code that is initially generated for a square loop uses a doubly-nested loop with two induction variables. These variables are incremented by multiplication, as the loop progresses through the array elements in each row, and at the start of each new row, as shown in [Figure 5.6](#).

Figure 5.6 Square loop



The loop transformation changes such a two-dimensional array into one row containing all the elements in one straight string. The multiplication instructions are replaced by additions, as the progression can now be performed on a linear basis. An example of the transformation of a square loop is shown below in [Figure 5.7](#).

Figure 5.7 Loop transformation - square loop

Optimization Techniques and Hints

Target-Independent Optimizations

C source code

```
int table[70][70];
int i, j;
for(i=0; i<35; i++)
    for(j=0; j<70; j++)
        c+=table[i][j];
```

Pseudocode before optimization

```
    i = 0;
L1  j = 0;
L2  tmp1 = i * 140;
    tmp2 = j * 2;
    tmp3 = tmp1 + tmp2;
    tmp4 = table[tmp3];
    c = c + tmp4;
    j++;
    if(j < 70) goto L2
    i++;
    if(i<35) goto L1
```

Pseudocode after optimization

```
...
tmp2 = table
Repeat 2450 times
    tmp4 = *tmp2
    c = c + tmp4
    tmp2 = tmp2 + 2
...
```

Assembly code output

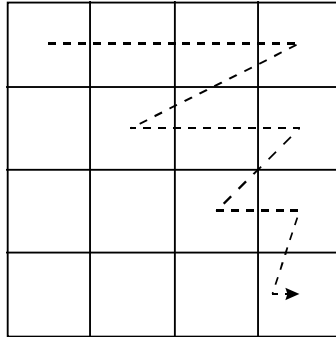
```
...
suba    r0,r0
move.l   #_tab,r1
move.w   #2450,d2
nop

doen3    d2
doestup3 L9
loopstart3
L9
move.w   (r1),r2
nop
adda     r2,r0
adda     #<2,r1
loopend3
...
```

Triangular loops

A triangular loop array access pattern is similar to the square loop described above, except that the pointer moves to an incremented starting position in each row. The starting position pointer increments by linear progression, as shown in [Figure 5.8](#):

Figure 5.8 Triangular loop



A triangular loop is transformed into a mainly linear based loop, incorporating the offset increment as an addition operation. [Figure 5.9](#) illustrates the transformation of a triangular loop.

Figure 5.9 Loop transformation - triangular loop

C source code

```
int table[70][70];
int i, j;
for(i=0; i<70; i++)
    for(j=i+3; j<70; j++)
        table[i][j] = 0;
```

Optimization Techniques and Hints

Target-Independent Optimizations

Pseudocode before optimization

```
i = 0;
L1 j = i
    if(j>=70) goto L3
L2     tmp1 = i * 140;
        tmp2 = j * 2;
        tmp3 = tmp1 + tmp2;
        table[tmp3] = 0;
        j++;
        if(j < 70) goto L2
L3 i++;
    if(i<70) goto L1
```

Pseudocode after optimization

```
...
tmp7 = 3 /* equal j+3, that is, */
        /* inner-loop low bound */
tmp4 = table + 6 /* pointer */
tmp6 = 8 /* reset to reach diagonal */
        /* after inner loop */
Repeat 70 times
    tmp5 = 70-tmp7
    if(tmp5<=0) goto L3
    Repeat tmp5 times
        *tmp4 = 0
        tmp4 = tmp4 + 2
    /* pointer prepared, set to */
    /* diagonal, next row */
L3 tmp4 = tmp4 + tmp6
    /* next step prepared */
    tmp6 = tmp6 + 2
    /* next number of iteration */
    /* for inner loop */
    tmp7 = tmp7 + 1
...

```

Assembly code output

```
...
    move.w    #<3,d0
    move.l    #_tab+6,r0
    move.w    #<8,r1
    move.w    #70,d5
    nop
    doen2     d5
    dosetup2  L10
    move.w    #70,d2
    loopstart2
L10
    sub       d0,d2,d3
    tstgt     d3
    jf        L4
    doensh3   d3
    clr       d6
    nop
    loopstart3
L9
    move.w    d6,(r0)
    adda      #<2,r0
    loopend3
L6
L4
    adda      r1,r0
    inc       d0
    adda      #<2,r1
    loopend2
...
```

Function inlining

Inlining replaces a call to a function with a copy of the code for the function. In cases where the procedure call and return may be more time-consuming than the function itself, function inlining can significantly increase the speed of the program. Function inlining generates larger executable code.

The function inlining optimization can be particularly effective with cross-file optimization, as it can be applied across all available files, and operates in conjunction with other cross-file optimizations.

[Figure 5.10](#) shows how the operation executed by the function Check is incorporated into the code itself, removing the call to the function.

Figure 5.10 Function inlining

Before optimization

```
int Check(int x);
{
    return (x>10);
}
void main()
{
    if (Check(y))
        a = 5;
}
```

After optimization

```
void main()
{
    if (y>10)
        a = 5;
}
```

You can force or suppress function inlining at specific points in the code, using the pragmas `#pragma inline` and `#pragma noinline`.

Common subexpression elimination

Where an expression appears in more than one place in the code and has the same computed value in each instance, this optimization replaces the expression itself with its result. Values loaded from memory can be included in this process, as well as values based on arithmetic computations. In [Figure 5.11](#), the variable `x` replaces the repeated subexpression `e + f`.

Figure 5.11 Common subexpression elimination

Before optimization

```
d = e + f + g;
y = e + f + z;
```

After optimization

```
x = e + f;
d = x + g;
y = x + z;
```

Loop invariant code

The term “invariant code” refers to an instruction which appears inside a loop, but whose value is not directly affected by the execution of the loop. This optimization moves such an instruction to a position outside the loop, with the result that the instruction is not repeated each time the loop executes. In [Figure 5.12](#), the variable `z` is set to the computed value of `2 * b + 1` before the loop executes, and this calculation is removed from the iteration.

Figure 5.12 Loop invariant code motion

Before optimization

```
b = c;
for (i=0; i<3; i++)
    d[i] = 2 * b + 1;
```

After optimization

```
b = c;
z = 2 * b + 1;
for (i=0; i<3; i++)
    d[i] = z;
```

Constant folding and propagation

This optimization identifies expressions which contain `int` values known to be constants and calculates their value at compilation time. The value of the expression then replaces the expression itself, as shown in [Figure 5.13](#) below.

Figure 5.13 Constant folding and propagation

Before optimization

```
X = 2;
Y = X + 10;
Z = 2 * Y;
```

After optimization

```
X = 2;
Y = 12;
Z = 24;
```

Jump-to-jump elimination

This optimization combines two jump operations into one, in cases where the code executes a jump to an address, and at that address immediately jumps to a different address.

In [Figure 5.14](#), the two jump instructions `goto J1;` and `goto J2;` are replaced by a direct jump to `J2`.

Figure 5.14 Jump-to-jump elimination

Before optimization

```
if (x)
    ...
else
    goto J1;

J1:
    goto J2;
```

After optimization

```
if (x)
    ...
else
    goto J2;
```

Dead code elimination

This optimization removes segments of “dead” code, meaning code that cannot possibly be executed. The code may be dead from the start, or it may become dead as a result of other optimizations. For example, the code may specify a condition which can never be true. In [Figure 5.15](#), the variable `c` is type `char`, which can never have a value greater than 255, and therefore the `if` condition will never be met.

Figure 5.15 Dead code elimination

Before optimization

```
char c;  
if c > 300  
    a = 1;  
else  
    a = 2;
```

After optimization

```
a = 2;
```

Dead storage/assignment elimination

Dead storage or assignment occurs when a variable is assigned a value, either directly or as a result of an expression, and is not used again anywhere in the code, or receives another value before being used. This optimization removes any unnecessary instructions and unused memory locations which may result from such cases. This redundancy may arise as a result of other optimizations.

In [Figure 5.16](#), before optimization the variable `a` is assigned the value 5, and is not used before it is reassigned the value 7. The dead storage/assignment elimination optimization removes the redundant instruction `a = 5`. If the variable `a` was not used at all after being assigned a value, it would be removed completely.

Figure 5.16 Dead storage/assignment elimination

Before optimization

```
a = 5;  
..  
a = 7;
```

After optimization

```
a = 7;
```

Target-Specific Optimizations

The Low-Level Transformations (LLT) phase is a separate modular stage of the optimization process which implements a number of target-specific optimizations. This phase transforms the linear code generated by the target-independent optimization phase into parallel assembly code, which can take advantage of the parallel execution units of the SC100 architecture.

The degree of parallelization that the optimizer is able to achieve is limited by the number and type of dependencies within the source code.

All target-specific optimizations are applied when the Level 2 optimization (option -O2) is selected. Target-specific optimizations are not activated at all when either option -O0 or option -O1 is selected.

The major target-specific optimizations are summarized in [Table 5.4](#), and examples of each are given in the sections that follow.

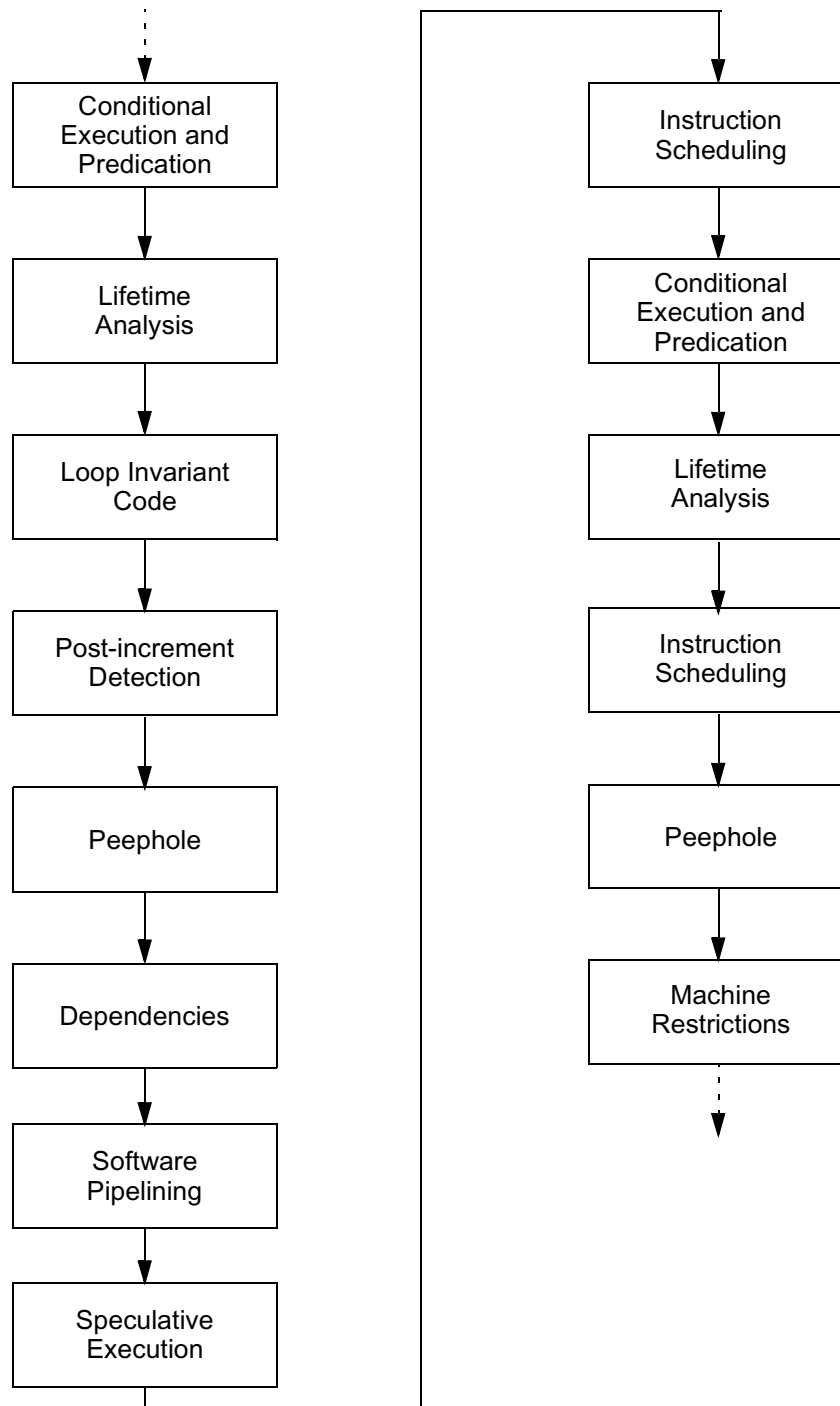
Table 5.4 Summary of target-specific optimizations

Optimization	Description
Instruction scheduling	Executes multiple instructions in the same cycle, fills delay slots associated with a branch operation, and avoids pipeline restrictions
Target-specific software pipelining	Rearranges instructions in a loop to minimize dependencies
Conditional execution and predication	Transforms a branch into a sequence of conditional actions
Speculative execution	Moves instructions from conditional to unconditional paths
Post-increment detection	Combines the functions of incrementing (or decrementing) a pointer and accessing the computed address into one instruction
Target-specific peephole optimization	Merges a sequence of instructions into a single instruction

The optimizer applies the target-specific optimizations in a predefined sequence, and invokes some of the optimizations more

than once, as illustrated in [Figure 5.17](#). Each optimization is directly affected by the result of the preceding optimization.

Figure 5.17 Sequence of target-specific transformation optimizations



Instruction scheduling

The main purpose of this optimization is to execute as many instructions as possible from the same instruction stream in the same cycle. The amount of dependency between the instructions limits the extent to which this can be achieved.

The instruction scheduling optimization organizes instructions into execution sets wherever it is possible to do so, making best use of the Data Arithmetic Units and Address Generation Units provided by the SC100 architecture.

[Figure 5.18](#) illustrates the use of instruction scheduling:

Figure 5.18 Instruction scheduling

<u>Before optimization</u>	<u>After optimization</u>
move.l d0, (r0)	move.l d0, (r0) inc d0
inc d0	tfra r3, r0 move (r1) +, d1
tfra r3, r0	adda #12, r3
adda #12, r3	
move (r1) +, d1	

Instruction scheduling serves two further purposes:

- Filling delay slots when branch instructions are executed
- Rescheduling operations that are not dependent on pipeline-restricted instructions

Filling delay slots

A branch instruction requires three cycles to execute if the branch is taken. When a branch executes, the prefetch queue is lost, and the cycles used for the other instructions are wasted, since they cannot execute until the branch instruction has completed. The wasted cycles are termed *delay slots*.

The instruction scheduling optimization checks whether other operations can be executed at the same time as the branch instruction. This is not possible if there are limiting factors, for example:

- The branch instruction is directly affected by the instructions which precede it.
- There are specific dependencies between the branch and the other instructions.

If there are no limiting factors, the scheduler rearranges the code, in order to use the delay slots efficiently. In [Figure 5.19](#), the code has been reorganized to enable three instructions to execute during the time that the branch requires to complete its operation.

Figure 5.19 Filling delay slots

<u>Before optimization</u>	<u>After optimization</u>
move.l d0, (r0)	rtsd
inc d0	move.l d0, (r0)
tfr d5, d2	inc d0
rts	tfr d5, d2

Avoiding pipeline restrictions

Certain instructions, for example, a move to an Rn register, are subject to pipeline restrictions. The effect of these instructions may not be implemented until two or more cycles after the instruction executes. In such cases, an operation which is dependent on the result of such an instruction, and which follows it immediately, must wait until the result is available.

The instruction scheduling optimization rearranges the sequence of such instructions where possible, using the cycle(s) which would otherwise be wasted to implement one or more operations that are not dependent on the restricted instruction.

In [Figure 5.20](#), the `clr` instruction has been rescheduled, since it can execute before the effect of the `move.l` instruction is implemented, whereas the `move.w` instruction must wait for the results of the `move.l` operation.

Figure 5.20 Avoiding pipeline restrictions

<u>Before optimization</u>	<u>After optimization</u>
move.l d0, (r0)	move.l d0, (r0)
nop	clr d0
move.w (r0), d1	move.w (r0), d1
clr d0	

Target-specific software pipelining

Software pipelining provides a further level of loop optimization, in addition to the target-independent optimizations which operate on loops.

The software pipelining optimization attempts to rearrange the sequence of instructions inside a loop, in order to minimize dependencies between such instructions, and thus increase the level of parallelization.

For example, a segment of code may consist of three instructions, A, B and C, within a loop which iterates 4 times. In some cases, the code may be reorganized into a different sequence without affecting its result, for example:

1. Instruction A
2. Instructions B, C, A, in a loop which iterates 3 times
3. Instruction B
4. Instruction C

The revised arrangement of the instructions results in fewer dependencies than in the original code.

This optimization is applied only to innermost loops of small or moderate size, which contain no branches or function calls within the loop. It is most effective when applied to loops that execute a large number of times.

Each iteration of a software pipelined loop may contain instructions from a different iteration of the original loop.

Software pipelining increases code size in almost all circumstances. When optimization for size is specified, software pipelining is suppressed entirely.

[Figure 5.21](#) shows how the software pipelining optimization reduces the number of iterations and rearranges instructions both within and outside the loop, thus enabling the maximum number of instructions that are not dependent on each other to execute in parallel.

Figure 5.21 Software pipelining - complex FIR

C source code

```
for (i = 0; i < N; i++)
{
L_tmpr = L_mac (L_tmpr, sample[i].r, coeff[N - i - 1].r;
L_tmpr = L_msu (L_tmpr, sample[i].i, coeff[N - i - 1].i;

L_tmpr = L_mac (L_tmpr, sample[i].i, coeff[N - i - 1].r;
L_tmpr = L_mac (L_tmpr, sample[i].r, coeff[N - i - 1].i;
}
```

<u>Before optimization</u>	<u>After optimization</u>
loop n times:	/* Prolog */
move.w (r0)+,d4	move.w (r0)+,d4 move.w (r1)-,d3
move.w (r1)-,d3	mac d3,d4,d5 move.w (r0)+,d1 move.w(r1)-,d2
mac d3,d4,d5	
move.w (r0)+,d1	loop n-1 times:
move.w (r1)-,d2	/*start loop*/
mac -d1,d2,d5	[
mac d3,d1,d6	mac d3,d1,d6 mac -d1,d2,d5
mac d2,d4,d6	move.w (r0)+,d4 move.w (r1)-,d3
]
	[
	mac d3,d4,d5 mac d2,d4,d6
	move.w (r0)+,d1 move.w (r1)-,d2
]
	/*endloop*/
	/* Epilog */
	mac d3,d1,d6 mac -d1,d2,d5
	mac d2,d4,d6

In [Figure 5.22](#), the loop iterates only 8 times, instead of the 10 in the original code, since two iterations have been unrolled. The loop executes in a single cycle. During this cycle the loop:

- Loads a value from iteration $i+2$
- Multiplies the value from iteration $i+1$
- Stores the result value from iteration i

Figure 5.22 Software pipelining - vector multiplication by a constant

C source code

```
for (i=0; i<10; i++)
    b[i] = mult(a[i], 0x4000);
```

Assembly code after optimization

```
doensh3    #<8 ; Pipelining loop twice
move.l     #_a,r1
move.f     #16384,d1
move.f     (r1)+,d0    move.l #_b,r0
mpy        d0,d1,d2    move.f (r1)+,d0
loopstart3

L93
[
    moves.f  d2,(r0)+    ; *
    mpy      d0,d1,d2    ; **
    move.f   (r1)+,d0    ; ***
]
loopend3

L92
moves.f    d2,(r0)+    mpy d0,d1,d2
moves.f    d2,(r0)+
```

Conditional execution and predication

The conditional execution and predication optimization simplifies small conditional structures and transforms the branch into one sequence.

An example of this transformation is shown in [Figure 5.23](#), in which two branches are removed.

Figure 5.23 Conditional execution and predication

C source code

```
If (a < 0) {  
    lower_bound = 0;  
    i = 0;  
}else  
    lower_bound = a;
```

Generated code before optimization

```
        move.w    a,d0  
        tstgt     d0  
        bf        L_False  
        clr       d2  
        clr       d3  
        bra       L_AfterIf  
L_False  
        tfr       d0,d2  
L_AfterIf  
        move.w    d2,lower_bound
```

Generated code after optimization

```
        move.w    a,d0  
        tstgt     d0  
        ift       clr d2      clr d3  
        iff       tfr d0,d2  
        move.w    d2,lower_bound
```

An additional advantage of this optimization is that it increases the size of the basic blocks in the optimized code segment, making further optimization more effective.

It is important to note, however, that the conditional execution optimization adds one word for each branch that it replaces (*ift* and *iff* in the above example). As a result, the impact on the size of the program can be considerable. Generally, this optimization is only activated for small structures where the number of instructions added is less or equal to the number of instructions saved. The optimization levels which specify size as an important consideration apply specific thresholds for this optimization.

Speculative execution

The speculative execution optimization moves instructions from conditional to unconditional paths, in order to fill execution slots that would not otherwise be used.

If an empty execution slot is available when a condition statement is encountered, the instructions are rearranged so that the conditional instructions execute unconditionally in previous cycles to the condition. If the condition is true and the *ift* instruction has been executed, or if the condition is false and the *iff* instruction has been executed, a cycle has been gained. If the condition result does

not match the moved instruction, the appropriate instruction is executed as normal, with no loss of cycles.

[Figure 5.24](#) shows an example of this transformation. In this example, the first `iff` instruction is moved so that it executes in the same cycle as the `cmpgt` instruction. If the result of the conditional operation is true, the `ift` instruction is executed in the next cycle. If the result is false, the instruction that was previously the second `iff` is executed, with the result that only one cycle is used instead of two.

Figure 5.24 Speculative execution

C source code

```
If (var > 5)
    x[3] = a;
else
    y = b;
```

Generated code before optimization

```
cmpgt #5,d1
nop
iff    move.l x+6,r0
iff    move.l d3,_y
ift    move.l d2,(r0)
```

Generated code after optimization

```
move.l x+6,r0    cmpgt #5,d1
nop
iff    move.l d3,_y
ift    move.l d2,(r0)
```

This optimization can be implemented successfully for one or more instructions if:

- Sufficient slots are available.
- There are no dependencies between the instruction in the conditional path and other instructions.
- The conditional instruction does not have any specific side effects.

Post-increment detection

This optimization exploits the features of the SC100 architecture, and increases code efficiency in terms of both size and speed. It identifies the instructions which use arithmetic functions to modify pointers, and which access the computed addresses, and replaces them with special post-increment or post-decrement address mode instructions which combine both functions.

The increment (or decrement) factor is not limited to the values 2 or 4, since any one of the four index registers (n0 through n3) may be used, as illustrated in [Figure 5.25](#).

Figure 5.25 Post-increment detection

<u>Generated code before optimization</u>	<u>Generated code after optimization</u>
<pre> L150 move.l #_L_R, r4 move.l #_CGUpdates, r5 doen3 #<8 dosetup3 L183 loopstart3 L183 move.l (r4), d0 move.l (r5), d1 mac d0, d1, d2 adda #<4, r4 adda #<12, r5 loopend3 L152 </pre>	<pre> L150 doensh3 #<7 ; Pipelining loop once move.w #3, n3 move.l #_L_R, r4 move.l #_CGUpdates, r5 move.l (r4)+, d0 move.l (r5)+n3, d1 loopstart3 L183 [mac d0, d1, d2 ; * move.l (r5)+n3, d1 ; ** move.l (r4)+, d0 ; **] loopend3 L152 mac d0, d1, d2 </pre>

Target-specific peephole optimization

The target-specific peephole optimization identifies sequences of instructions that can be merged into a single instruction, and implements this transformation, as shown in [Figure 5.26](#).

Figure 5.26 Target-specific peephole optimization

<u>Generated code before optimization</u>	<u>Generated code after optimization</u>
<pre> deca r0 move.w #33, d0 tstgea.l r0 </pre>	<pre> decgea r0 move.w #33, d0 </pre>

[Figure 5.27](#) illustrates a combination of pipelining and peephole optimizations. After pipelining, the final mac instruction, which has been moved outside the loop, is merged with the rnd instruction to form a macr instruction.

Figure 5.27 Combined pipelining and peephole optimizations

<u>Generated code before optimization</u>		<u>Generated code after optimization</u>	
	doen #9	doen #8	; Pipelining loop once
	dosetup0 L1	dosetup0 L1	
	loopstart0	move.w (r0)+,d3	
L1	move.w (r1)+,d2	move.w (r1)+,d2	
	mac d2,d3,d7	loopstart0	
	loopend0	mac d2,d3,d7	; *
		move.w (r1)+,d2	; **
		move.w (r0)+,d3	; **
		loopend0	
	rnd d7		
		macr d2,d3,d7	

Prefix grouping

Instruction grouping is applied by the optimizer wherever possible, in order to make best use of the available multiple execution units. In addition to “natural” grouping of instructions, which increases efficiency and does not increase code size, the optimizer can implement prefix grouping. Prefix grouping is a mechanism whereby an additional word is introduced into the code in order to force more than one instruction to execute in the same cycle.

Prefix grouping improves performance in terms of speed, but increases the size of the code. The optimizer activates prefix grouping on the entire code.

Space Optimizations

When you select the `-Os` option, the optimizer aims to produce code that occupies as little memory space as possible for the given optimization level. In certain cases, the reduced memory space may be at the expense of program speed.

The compiler executes all optimizations associated with the specified optimization level, except for those that adversely affect code size, as noted below:

- For target-independent optimizations, `-Os` disables function inlining as this always increases code size.
- For target-specific optimizations, `-Os` does the following:
 - Disables software pipelining.

- Omits conditional execution if the basic block involved contains more than five instructions.
- Uses only serial grouping when encoding assembly instructions, since code size is increased when prefixes are added.

The optimizer applies conditional execution and predication to small structures only, because this optimization adds to the size of the code.

The `-Os` option may be used in combination with any other optimization option except `-O0`. If no optimization level is specified with `-Os`, Level 2 optimization (`-O2` option) is selected by default.

The command line shown in [Listing 5.6](#) invokes the optimizer with the default Level 2 optimizations. All target-independent and target-specific optimizations, except those noted above, are applied across all modules in the application.

Listing 5.6 Invoking the optimizer for space optimization

```
scc -Os -Og -o file1.eld file1.c file2.c
```

Cross-File Optimizations

Cross-file optimization produces the most effective form of optimization, since optimizations are applied across all the files in the application. The option `-Og` can be specified in the command line together with any of the optimization options except the `-O0` option, and is most effective when used with the default level `-O2`.

In addition to implementing the selected level of optimization across all the files, cross-file optimization also applies two specific optimizations:

- Function inlining across multiple files; this applies function inlining to the whole program. As with function inlining for individual files, this increases the size of the code, but can considerably increase execution speed.
- Optimization of access to global and static variables.

Guidelines for Using the Optimizer

The optimizer produces the best possible results when the source code is written in a simple and straightforward manner. Complex structures and algorithms should be avoided wherever possible, since these can reduce the effectiveness of many of the optimizations.

During the various optimization phases, the compiler attempts to convert all the structures in the code into a form that is independent of the style of an individual user, and that can be processed efficiently by the individual optimizations. By following the basic rules of clarity and simplicity when writing your code, you help the optimizer to retrieve the specific information it needs, and to apply the maximum amount of optimization.

For example, when accessing arrays you should use simple access instructions wherever possible, and avoid using complex access instructions which use pointers, as shown in [Listing 5.7](#):

Listing 5.7 Simple and complex array accesses

```
# Simple array access (recommended)
a[i];

# Complex array access (not recommended)
p = &a[0]
*p++;
```

You can further enhance the results of the optimization by applying two specific techniques that help the optimizer take full advantage of the multiple execution units of the SC100 architecture:

- Partial summation, which reduces dependencies in a loop, enabling multiple iterations of a loop in parallel
- Multisample processing, a programming technique which processes multiple samples simultaneously

These techniques are described in the sections that follow.

Partial Summation Techniques

One of the optimizer's major functions is to produce parallelized code that fully utilizes the available number of multiply-accumulate

Optimization Techniques and Hints

Partial Summation Techniques

(MAC) units. The number of MAC units that can be used in an execution set, meaning the number of instructions executed in the same cycle, is usually limited by the degree of dependency within the code.

The partial summation programming technique helps you reduce the dependencies in the loops in your source code, in such a way that the iterations can execute in parallel. By structuring your source code using partial summation techniques wherever possible, you enable the optimizer to further reduce dependencies and increase parallelization.

In [Figure 5.28](#), the inner loop can use only a single MAC per cycle, because of the inner dependency within the algorithm. The same output code is generated when compiling for a single, dual, or quad MAC StarCore® system.

Figure 5.28 MAC usage limited by dependency in loop

Source code

```
void Iir(short Input[], short Coef[], short FiltOut[])
{
    long L_Sum = 0;  short int Stage, Smp;  int LoopCount;

    FiltOut[0] = Input[0];
    for (Smp = 1; Smp < S_LEN; Smp++)
    {
        L_Sum = LPC_ROUND;  LoopCount = (Smp < NP ? Smp : NP );

        for (Stage = 0; Stage < LoopCount; Stage++)
            L_Sum = L_msu(L_Sum, FiltOut[Smp - Stage - 1], Coef[Stage]);

        L_Sum = L_shl(L_Sum, ASHIFT);
        L_Sum = L_msu(L_Sum, Input[Smp], 0x8000);
        FiltOut[Smp] = extract_h(L_Sum);
    }
}
```

Generated code

```
doenshl d0
move.f r2)+,d0      move.f (r0)-,d1
loopstart1
PL001
mac    -d0,d1,d2      move.f (r0)-,d1      move.f (r2)+,d0
loopend1
PL000
mac    -d0,d1,d2
```

[Figure 5.29](#) illustrates how you can use partial summation to split the inner loop in the above example to enable two parallel iterations. The loop iterates half the number of times. The sum is accumulated using two variables, which are combined outside the loop.

Figure 5.29 Partial summation for dual MAC usage

Source code

```
for (Stage = 0; Stage < (LoopCount>>1); Stage++)
{
    L_Sum = L_msu(L_Sum, FiltOut[Smp - 2*Stage -1], Coef[2*Stage]);
    L_Sum1 = L_msu(L_Sum1, FiltOut[Smp - 2*Stage -2], Coef[2*Stage+1]);
}

L_Sum = L_shl(L_Sum+L_Sum1, ASHIFT);
L_Sum = L_msu(L_Sum, Input[Smp], 0x8000);
```

Generated code

```
doenshl d0
move.2f (r2)+,d0d1    move.2f (r0)-,d6d7
loopstart0
PL001
[
mac    -d0,d6,d2
mac    -d1,d7,d5
move.2f (r0)-,d6d7
move.2f (r2)+,d0d1
]
loopend0
PL000
mac    -d0,d6,d2      mac    -d1,d7,d5
```

The same technique can be used for compiling with a quad MAC system, by splitting the loop into four iterations, using four variables and one quarter the number of iterations.

It is important to note that partial summation is not suitable for algorithms with bit-exact requirements. This technique changes the order of the calculation, and may affect the value of the result in cases where statements must be executed in the exact order specified.

In certain algorithms the effectiveness of the partial summation technique may be limited because of alignment restrictions. For example, the `move.2f` instruction, which is required for partial summation, must be used on a long word boundary.

In [Listing 5.8](#), this restriction is satisfied, and the partial summation technique can be used successfully. [Listing 5.8](#) shows an algorithm for which partial summation cannot be used. This is because the second iteration produces an odd value for the variable `i`, with the result that the `move.2f` instruction violates the alignment requirement.

Listing 5.8 Alignment restrictions in algorithms

```
for (i = 0; i < DataBlockSize; i++)
{
    Delay[(DataBlockSize-i)] = DataIn[i];
    sum1 = 0; sum2 = 0;
    for (j = 0; j < FirSize/2 ; j++)
    {
        sum = L_mac(sum, Coef[2*j], Delay[2*j-i]);
        sum = L_mac(sum, Coef[2*j+1], Delay[2*j-i+1]);
    }
    Result = round(sum);
}
```

The multisample techniques described in the following section help you write source code which enables the optimizer to take further advantage of multiple execution units. You can apply multisample techniques even if you cannot use partial summation for certain algorithms because of alignment restrictions or bit-exact requirements.

Multisample Techniques

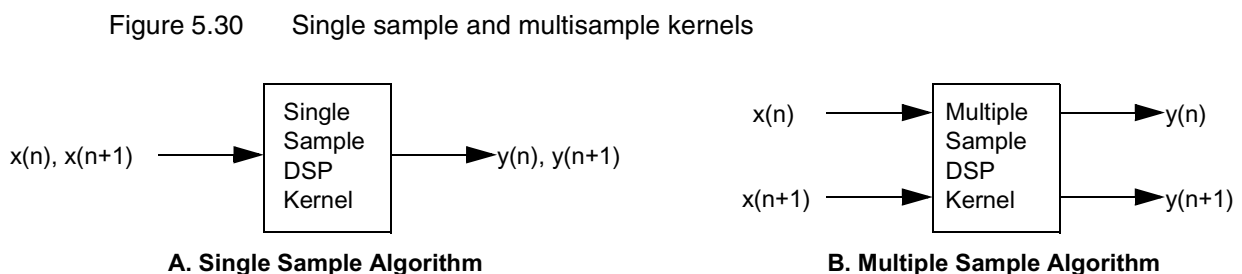
To obtain high performance, a pipelining technique called “multisample” programming is used to process multiple samples simultaneously. The multisample programming techniques enable you to obtain high performance by taking full advantage of the SC100 multiple-ALU architecture.

The following terminology is used throughout this section:

- **Generic Kernel:** The minimum required operations of the algorithm. The generic kernel is the theoretical minimum size of the kernel without considering implementation constraints.
- **Basic Kernel:** The inner loop of a DSP algorithm. This may contain several replications of the generic kernel or additional code for pipelining. The basic kernel is actually implemented on the DSP and is subject to implementation constraints.
- **Operand:** A value used as an input to an ALU.
- **Delays:** Values stored as a delay line for referencing past values.
- **Iteration:** The complete execution of a basic kernel.
- **Loop pass:** The execution of the instructions within the basic kernel. Many loop passes may be needed to complete a single iteration of the kernel.

To process several samples simultaneously, operands (both coefficients and variables) are reused within the kernel. Although a coefficient or operand is loaded once from memory, multiple ALUs may use the value, or the value may be used in a later step of the kernel.

[Figure 5.30](#) illustrates the structure of a single sample and multisample algorithm.



In a single sample algorithm ([Figure 5.30 A](#)), samples are processed by the algorithm serially. The kernel processes a single input sample

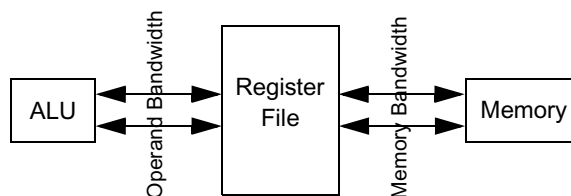
and generates a single output sample. For an algorithm such as an FIR, samples are input to the FIR kernel one at a time. The FIR kernel generates a single output for each input sample. Blocks of samples are processed using loops and executing the FIR kernel several times.

In contrast, the multisample algorithm ([Figure 5.30 B](#)) takes multiple samples at the input in parallel and generates multiple samples at the output simultaneously. The multisample algorithm operates on data in small blocks. Operands and coefficients are held in registers, and applied to both samples simultaneously, resulting in fewer memory accesses.

Multisample algorithms are ideal for block processing algorithms where data is buffered and processed in groups (such as speech coders). [Figure 5.30 B](#) shows two samples being processed simultaneously. However, the number of simultaneous samples depends on the processor architecture and type of algorithm.

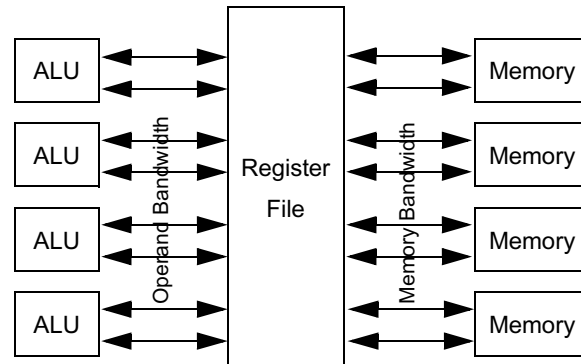
Most DSP algorithms have a multiply-accumulate (MAC) at their core. On a load/store machine, the register file is the source/destination of operands to/from memory. For the ALU, the register file is the source/destination of operands. On a single sample, single ALU algorithm, the memory bandwidth is typically equal to the operand bandwidth, as shown in [Figure 5.31](#).

Figure 5.31 Single ALU operand and memory bandwidth



When increasing the number of ALUs to four, the bandwidth increases as shown in [Figure 5.32](#).

Figure 5.32 Quad ALU operand and memory bandwidth



Quadrupling the number of ALUs quadruples the operand bandwidth. If there is one address generator per operand, this results in eight address generators. This is undesirable because it requires an 8 port memory and a significant amount of address generation hardware.

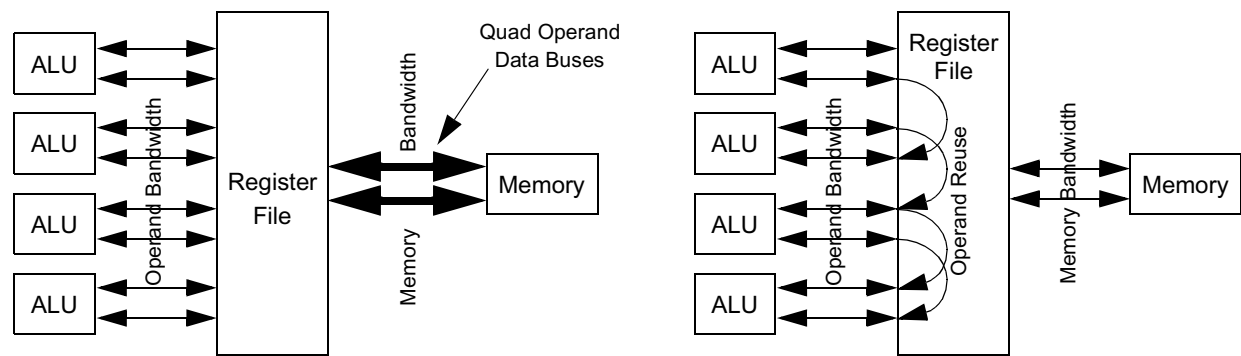
The SC140 DSP core solves this problem by providing up to a quad operand load/store over a single bus. With two quad operand loads, eight operands can be loaded using two address generators.

Although quad operand loading provides the proper memory bandwidth, some algorithms have special memory alignment requirements. These alignment requirements make it difficult to use multiple operand load/stores.

Multisample algorithms provide a solution for implementing algorithms with memory alignment requirements. By reusing previously loaded values, the number of operands loaded from memory is reduced, which relaxes the alignment constraints.

Both techniques for increasing operand bandwidth, by using wider data buses or by reusing operands, are shown in [Figure 5.33](#).

Figure 5.33 Options for increasing operand bandwidth



To introduce the multisample technique, four example DSP kernels are written in multisample form. The DSP kernels presented are direct form FIR filter, direct form IIR filter, correlation and biquad filter.

Multisample implementation issues

When implementing a DSP algorithm such as an FIR filter, trade-offs are made between the number of samples processed and the number of ALUs as shown in [Figure 5.34](#).

Figure 5.34 Number of samples and ALUs for implementing DSP algorithms

		Number of ALUs		
		1	2	4
Number of Samples	1	1 sample, 1 ALU	1 sample, 2 ALUs	1 sample, 4 ALUs
	2	2 samples, 1 ALU	2 samples, 2 ALUs	2 samples, 4 ALUs
	4	4 samples, 1 ALU	4 samples, 2 ALUs	4 samples, 4 ALUs

As the kernel computes more samples simultaneously, the number of memory loads decreases because data and coefficient values are being reused. However, to obtain this reuse, more intermediate results are required, which typically requires more registers in the processor architecture.

If the operand memory requires wait states, this technique improves the speed of the algorithm. If the operand memory is full speed,

then the algorithm does not execute any faster, but may reduce power consumption because the number of memory accesses has been reduced.

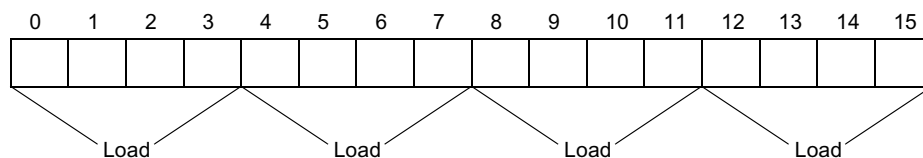
By using more ALUs, it is theoretically possible to compute an algorithm faster. Moving across the row theoretically applies 1, 2 or 4 ALUs to the algorithm. To apply multiple ALUs, some degree of parallelism is required in the algorithm to partition the computations.

Although computing a single sample with multiple ALUs is theoretically possible, limitations in the DSP hardware may not allow this style of algorithm to be implemented. In particular, most processors typically require operands to be aligned in memory and multiple operand load/stores to be aligned.

For example, a double operand load requires an even address and a quad operand load requires a double even address. These types of restrictions are typical to reduce the complexity of the address generation hardware (particularly for modulo addressing).

Restricting the boundaries of the load makes implementing some algorithms very difficult or impossible. This is easiest to explain by way of example. Consider a series of (aligned) quad operand loads from memory, as shown in [Figure 5.35](#).

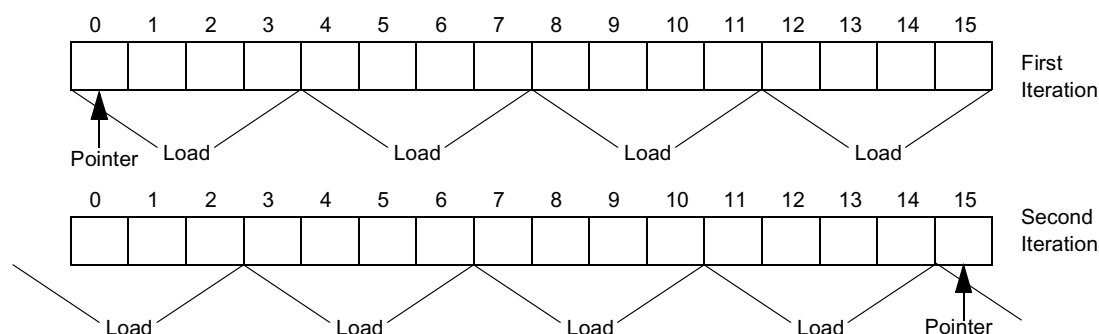
Figure 5.35 Quad coefficient loading from memory



The loads in [Figure 5.35](#) do not have a problem with alignment because loads occur from double even addresses.

Alignment problems typically occur with algorithms implementing delay lines in memory. These algorithms delete the oldest delay and replace it with the newest sample. This is typically done by using modulo addressing and “backing up” the pointer after the sample is processed. This leads to an addressing alignment problem as shown in [Figure 5.36](#).

Figure 5.36 Misalignment when loading quad operands



On the first iteration of the kernel, quad data values are loaded starting from a double even address. This does not create an alignment problem. However, at the end of the first iteration, the pointer is backed up by one to delete the oldest sample. On the next iteration, the pointer is not at a double even address and the quad data load is not aligned.

A solution to the alignment problem is to reduce the number of operands moved on each data bus. This relaxes the alignment issue. However, in order to maintain the same operand bandwidth, each loaded operand must be used multiple times. This is a situation where multisample processing is useful.

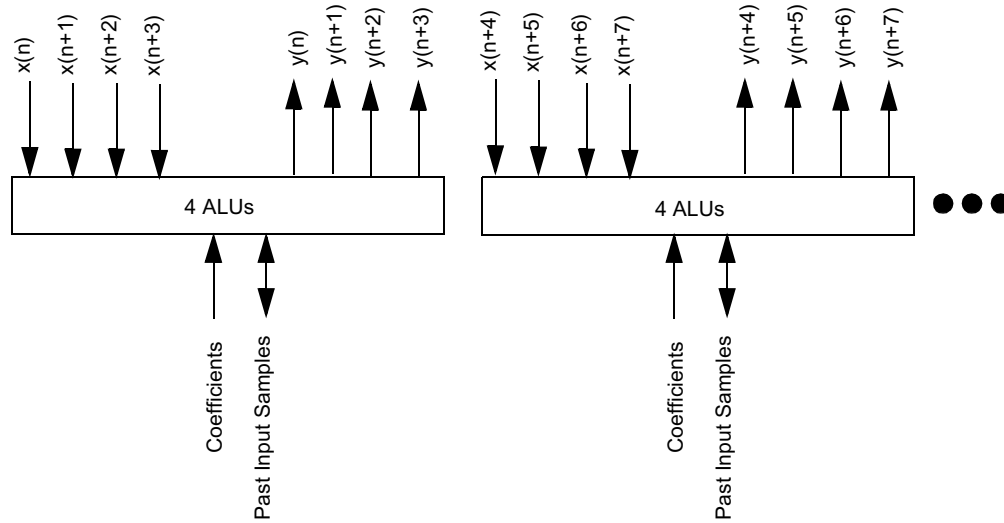
As the number of samples per iteration increases, more operands are reused and the number of moves per sample is reduced. With fewer moves per sample, the number of memory loads is decreased allowing fewer operands per bus. Fewer operands per bus allows the data to be loaded with fewer restrictions on alignment.

Implementation example

The FIR_A4S4 Quad ALU, quad sample, is the highest performance implementation on a quad ALU SC100 DSP.

To further increase the performance of the FIR filter, four ALUs may be used. To avoid misalignment, four samples are processed simultaneously. The quad ALU, quad sample FIR data flow is shown in [Figure 5.37](#).

Figure 5.37 Quad ALU, quad sample FIR filter data flow



Input samples are grouped together four at a time. Coefficients and delays are loaded and applied to all four input values to compute four output values. By using four ALUs, the execution time of the filter is only one quarter the execution time of a single ALU filter.

To develop the FIR filter equations for processing four samples simultaneously, the equations for the current sample $y(n)$ and the next three output samples $y(n+1)$, $y(n+2)$ and $y(n+3)$ are as shown in [Figure 5.38](#).

Figure 5.38 FIR filter equations for four samples

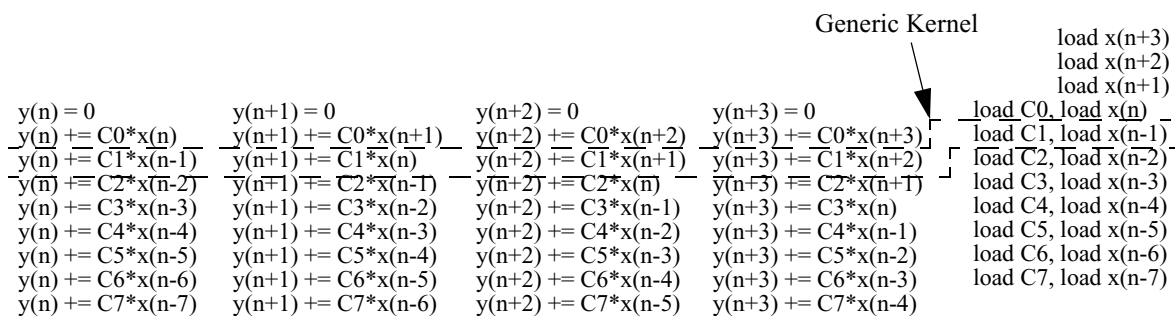
$$\begin{aligned}
 y(n) &= x(n)C_0 + x(n+1)C_1 + x(n+2)C_2 + x(n+3)C_3 + x(n+4)C_4 + x(n+5)C_5 + x(n+6)C_6 + x(n+7)C_7 \\
 y(n+1) &= x(n+1)C_0 + x(n+2)C_1 + x(n+3)C_2 + x(n+4)C_3 + x(n+5)C_4 + x(n+6)C_5 + x(n+7)C_6 + x(n+8)C_7 \\
 y(n+2) &= x(n+2)C_0 + x(n+3)C_1 + x(n+4)C_2 + x(n+5)C_3 + x(n+6)C_4 + x(n+7)C_5 + x(n+8)C_6 + x(n+9)C_7 \\
 y(n+3) &= x(n+3)C_0 + x(n+4)C_1 + x(n+5)C_2 + x(n+6)C_3 + x(n+7)C_4 + x(n+8)C_5 + x(n+9)C_6 + x(n+10)C_7
 \end{aligned}$$

The generic kernel has the following characteristics:

- Four parallel MACs.
- One coefficient is loaded and used by all four MACs in the same generic kernel.
- One delay value is loaded, used by the generic kernel and saved for the next three generic kernels.
- Three delays are reused from the previous generic kernel.

To develop the structure of the quad ALU kernel, the filter operations are written in parallel and the loads are moved ahead of where they are first used. This creates the generic kernel shown in [Figure 5.39](#).

Figure 5.39 Generic kernel for quad ALU FIR



The generic kernel requires four MACs and two parallel loads. [Figure 5.40](#) illustrates how the kernel in [Figure 5.39](#) is implemented in a single instruction.

Figure 5.40 Single instruction quad ALU generic filter kernel

$y(n) += C * D1$	$y(n+1) += C * D2$	$y(n+2) += C * D3$	$y(n+3) += C * D4$
Load C, Copy D3 to D4, Copy D2 to D3, Copy D1 to D2, Load D1			

To provide delay reuse, the delays are copied by using temporary variables D1, D2, D3 and D4 as a delay line. This imposes a requirement on the kernel to perform two MACs and five move operations (two loads and three copies) in a single instruction.

[Listing 5.9](#) contains an example of C simulation code which implements the generic kernel shown in [Figure 5.39 on page 152](#).

Listing 5.9 FIR_A4S4 quad ALU, quad sample C simulation code

```
#include <prototype.h>
#include <stdio.h>

#define DataBlockSize 40 // size of data block to process
#define FirSize      8  // number of coefficients in FIR

Word16 DataIn[DataBlockSize] = {
    328, 9830, 8192, -6553, -3277, 3277, 3277, -6553, -9830, 4915,
    8192, -6553, 328, 9830, 4915, -6553, -3277, 3277, 3277, -9830,
    4915, -3277, -9830, 8192, -6553, 328, 9830, -6553, 3277, 3277,
    3277, 328, 9830, 4915, -3277, -9830, 8192, -6553, -6553, 3277
};

Word16 Coef[FirSize] = {
    3277, 6553, -9830, -6553, -4915, 3277, 8192, -6553
};
Word16 Delay[FirSize+3];

#define IncMod(a) (a=((a+1)%(FirSize+3)))
#define DecMod(a) (a=((a+FirSize+2)%(FirSize+3)))

int main()
{
    int DelayPtr;
    Word32 sum1,sum2,sum3,sum4;
    Word16 D1,D2,D3,D4;
    int i,j;

    DelayPtr = 0; // init delay ptr

    for (i = 0; i < DataBlockSize; i += 4) {
        // do 4 samples at a time

        Delay[DelayPtr] = DataIn[i];   DecMod(DelayPtr);
        Delay[DelayPtr] = DataIn[i+1]; DecMod(DelayPtr);
        Delay[DelayPtr] = DataIn[i+2]; DecMod(DelayPtr);
        Delay[DelayPtr] = DataIn[i+3];

        sum1 = 0; // init sum to zero
        sum2 = 0; // init sum to zero
        sum3 = 0; // init sum to zero
```

Optimization Techniques and Hints

Multisample Techniques

```
sum4 = 0;          // init sum to zero

D4 = Delay[DelayPtr];    IncMod(DelayPtr);
D3 = Delay[DelayPtr];    IncMod(DelayPtr);
D2 = Delay[DelayPtr];    IncMod(DelayPtr);

for (j = 0; j < FirSize / 4 ; j++) { // evaluate FIR
    D1 = Delay[DelayPtr]; // get delay
    IncMod(DelayPtr);

        sum1 = L_mac ( sum1, Coef[4*j], D1 );
        sum2 = L_mac ( sum2, Coef[4*j], D2 );
        sum3 = L_mac ( sum3, Coef[4*j], D3 );
        sum4 = L_mac ( sum4, Coef[4*j], D4 );

    D4 = Delay[DelayPtr]; // get delay
    IncMod(DelayPtr);

        sum1 = L_mac ( sum1, Coef[4*j+1], D4 );
        sum2 = L_mac ( sum2, Coef[4*j+1], D1 );
        sum3 = L_mac ( sum3, Coef[4*j+1], D2 );
        sum4 = L_mac ( sum4, Coef[4*j+1], D3 );

    D3 = Delay[DelayPtr]; // get next delay
    IncMod(DelayPtr);

        sum1 = L_mac ( sum1, Coef[4*j+2], D3 );
        sum2 = L_mac ( sum2, Coef[4*j+2], D4 );
        sum3 = L_mac ( sum3, Coef[4*j+2], D1 );
        sum4 = L_mac ( sum4, Coef[4*j+2], D2 );

    D2 = Delay[DelayPtr]; // get next delay
    IncMod(DelayPtr);

        sum1 = L_mac ( sum1, Coef[4*j+3], D2 );
        sum2 = L_mac ( sum2, Coef[4*j+3], D3 );
        sum3 = L_mac ( sum3, Coef[4*j+3], D4 );
        sum4 = L_mac ( sum4, Coef[4*j+3], D1 );
    }
    DecMod(DelayPtr);

    printf("Index: %d, output: %d\n", i, round(sum1));
```

```
printf("Index: %d, output: %d\n", i+1, round(sum2));  
printf("Index: %d, output: %d\n", i+2, round(sum3));  
printf("Index: %d, output: %d\n", i+3, round(sum4));
```

General Hints

In addition to the specific techniques described in the previous sections, there are a number of general guidelines that you should follow when writing source code, in order to assist the optimizer to produce the most efficient results. These guidelines are described in the sections that follow.

Software pipelining

The optimizer implements sophisticated levels of software pipelining, saving you the need to introduce software pipelining into your source code. It is important that you do not include any manual form of software pipelining into your source code, as this can conflict with the algorithms used by the optimizer, resulting ultimately in less efficient optimization.

[Listing 5.10](#) shows two forms of source code for the same loop. The first version contains no pipelining, and is the recommended source code form. This will generate more efficient and smaller code than the second version, which pipelines the first iteration at the C level outside the loop. The type of manual pipelining shown in the second version should be avoided.

Listing 5.10 Avoiding software pipelining in source code

```
# 1. No pipelining (recommended)  
  
L_R = 0;  
  
for (J = 0; J < S_LEN; J++)  
    L_R = L_mac(L_R, WBasisVecs[J + (I * S_LEN)], WInput[J]);  
  
# 2. Manual pipelining (not recommended)  
  
L_R = L_mult(WBasisVecs[I * S_LEN], WInput[0]);
```

```
for (J = 1; J < S_LEN; J++)  
    L_R = L_mac(L_R, WBasisVecs[J + (I * S_LEN)], WInput[J]);
```

Passing and returning large structs

Instead of passing and returning large structs using their value, use pointers to large structs wherever possible.

Arithmetic operations

Whenever you can, use constants instead of variables for shift, division, or remainder operations.

Local variables

Any local variable that you specify should be initialized before it is used.

Resource limitations

The SC100 architecture provides a total of 16 Dn registers and 16 Rn registers. If the number of active variables is greater than the number that the registers can accommodate, the compiler maps the extra variables to memory, resulting in less efficient code.

For best results, you should take account of these physical limitations when writing your source code. For example, when preparing a set of instructions to execute in one cycle, remember that there is a restriction on the number of operands that can be used in a single cycle.

Optimizer Assumptions

The optimizer uses the information passed to it by the compiler, in order to ensure that the optimizations applied during the various optimization stages do not affect the original accuracy of the program.

At the time that the compiler accumulates this information, it assumes that only two types of variables can be accessed while inside a function, either indirectly through a pointer or by another function call:

- Global variables, meaning all variables within the file scope or application scope

- Local variables, whose addresses are retrieved implicitly by the automatic conversion of array references to pointers, or explicitly by the `&` operator

If your programs conform to the standard ANSI/ISO version of C, this assumption does not affect your code. If the code that you are compiling is not standard, and it violates this assumption, the optimization process may adversely affect the behavior of the program.

To avoid unexpected results, and to ensure that your program executes correctly once optimized, follow the coding guidelines listed below:

- Don't make assumptions based on memory layout when using pointers. For example, if `x` points to the first member of a structure, `x+1` may not necessarily point to the second member of the same structure. Similarly, if `y` is defined as a pointer to the first declared variable in a list, do not assume that `y+1` points to the second variable in the list.
- When referencing an array, keep the references inside the array bounds.
- Ensure that all the required arguments are passed to functions.
- When subscribing one array, don't access another array indirectly. For example, if in the construct `x[y-x]`, `x` and `y` are the same type of array, the construct is equivalent to `*(x+(y-x))`, which is equivalent to `*y`. Thus the construct actually references the array `y`.
- When pointing to objects, don't reference outside the bounds of these objects. The optimizer assumes that all references of the form `*(p+i)` apply within the bounds of the variable(s) to which `p` points.
- When the need arises for variables that are accessed by external processes, be sure to declare the variables as `volatile`. Use this keyword judiciously, as it may have adverse effects on optimization.

Optimization Techniques and Hints

Optimizer Assumptions

Runtime Environment

This chapter describes the startup code used by the Metrowerks Enterprise C compiler, the layout and configuration of memory, and the calling conventions which the compiler supports.

This chapter contains the following topics:

- [Startup Code](#)
- [Memory Models](#)
- [Memory Layout and Configuration](#)
- [Calling Conventions](#)

Startup Code

The compiler runtime startup code consists of the following components:

- Initialization code, which is executed when the program is initiated and before its main function is called
- Finalization code, which controls the closedown of the application after the program's main function terminates
- Entry points for low-level I/O services
- The interrupt vector table
- Support for debugging tools

For CodeWarrior for the StarCore DSP, the entire startup code for the compiler is contained in a single assembly code file, named `crtsc4.asm`, which is located in the following directory:

Windows	StarCore Support\Compiler\src\rtlib
Solaris	StarCore/starcore_support/src/rtlib

When the object module for this file is generated, the file is called `crtsc4.eln` and is located in the following directory:

Windows `StarCore Support\Compiler\lib`

Solaris `Starcore/starcore_support/lib`

The compiler startup code contains two phases:

- **Bare board startup code**, which is used for programs which execute without the support of any runtime executive or operating system. This phase resets the interrupt vector and initializes all necessary hardware registers.
- **C environment startup code**, which is a mandatory phase for all configurations. This phase initializes the runtime structure of the application for the C environment, and includes the finalization code used following termination of the program.

Bare Board Startup Code

The bare board startup phase assumes that no operating system or runtime executive is running. It performs the various actions which are normally carried out automatically by the operating system or runtime executive, as follows:

1. The reset interrupt vector is set to point to the system entry point `__crt0_start`, as if the system has just been reset. The interrupt vector table holds the addresses of all interrupt handlers. The first entry in this table is the system entry point. All other entries in the interrupt vector table point by default to the `abort` function.
2. The hardware registers are initialized as follows:
 - The four modulo (M) registers (`m0-m3`) are initialized to linear addressing.

- The status register is set to an initial value taken from the linker command file used at link time. This file includes a label `SR_setting`, which defines the initial value to be assigned to the status register following system reset. [Table 6.1](#) shows the default status register settings.

Table 6.1 Status register default settings

Setting Type	Value
Mode:	Exception mode
Interrupt level:	7
Saturation:	On
Rounding mode:	NEAREST_EVEN

3. If the system includes a timer, the timer is activated.
4. The bare board startup phase terminates by jumping to the C environment startup code entry point, `__start`.

C Environment Startup Code

The C environment startup phase is applicable to all programs. The entry point for this phase is `__start`. This phase includes initialization code used prior to program start, and finalization code used after the application terminates.

C environment initialization code

The following initialization actions are executed before the application starts:

1. The memory map is set up and initialized. The stack pointer (SP) value is loaded into memory by the stack start address, located at `StackStart`. This label is defined in the linker command file and used by the linker at link time.
2. If the `-mrom` option has been specified in the shell command line, initialized variables are copied from ROM into RAM. This option is required for applications which do not use a loader.
3. The `argv` and `argc` arguments are set up.
4. Interrupts are enabled. Until this point, interrupts have been disabled.

5. The application main procedure entry point is called using the function `main`.

Initialization of variables

If your system uses a loader, this will by default initialize all variables. In systems that do not include a loader, it is important that you specify the `-mrom` option when you compile the final version of your application, to ensure that the initialized variables are copied from ROM into RAM at startup.

NOTE Before a C program executes, certain global variables may assume the assignment of an initial value of zero. The compiler does not preinitialize variables automatically. You must ensure that your code includes explicit initialization of any variable that must have an initial value of zero.

C environment finalization code

On return from the application main function, the runtime function `exit` is called. This terminates any I/O services which have not yet terminated, and stops the processor by issuing the `stop` instruction.

NOTE Certain embedded real time applications never terminate. Such termination activities do not usually pertain to embedded applications, but may be of use during early development and debugging stages.

Low-level I/O services

The C environment startup code includes the input and output of low-level, buffered I/O services. The code uses calls to `__send` and `__receive` in order to interface with debugging tools and/or runtime systems.

Configuring Your Startup Code

If the default runtime setup does not match your configuration, you need to modify your startup code accordingly.

To create your own runtime configuration code, follow the steps described below:

- 1 Make your own copy of the default startup file, `crtsc4.asm`, with a name of your choice, as shown in [Listing 6.1](#):

Listing 6.1 Creating a new startup file

```
cp install-dir/src/rtlib/crtsc4.asm mysc100.asm
```

- 2 Make the required changes to the new file.
- 3 Assemble the modified file, as shown in [Listing 6.2](#).

Listing 6.2 Assembling the modified startup file

```
asmsc100 -b -l mysc100.asm
```

The generated object file has the same file name as the source file, and the extension `.eln`. In this example, the object file generated is `mysc100.eln`.

- 4 Use the modified file by specifying the `-crt` option in the shell command line, as shown in [Listing 6.3](#), to ensure that the modified startup file is used at link time.

Listing 6.3 Using the modified startup file

```
scc -crt mysc100.eln my-object-files.eln
```

Memory Models

The compiler architecture supports both 16-bit and 32-bit addresses. If the application is small enough to allow all static data to fit into the lower 64K of the address space, then more efficient code can be generated. This mode, the small memory model, is the default, and assumes that all addresses are 16-bit.

The big memory model does not restrict the amount of space allocated to addresses. This model is selected with the option `-mb` when the shell is invoked.

When the compiler uses the big memory model to access a data object, whether static or global, it must use a longer instruction that includes a 32-bit address. This operation requires an additional

word, and as a result it produces code that is larger, and in some cases, slower, than a similar operation using the small memory model.

[Listing 6.4](#) illustrates the use of the `move.l` instruction in the big and small memory models. In this example, the assembler interprets the address in the first instruction as a 32-bit address, and allocates the maximum space for it. In the second instruction, the `<` symbol indicates to the assembler that this address fits into a 16-bit space, thus preventing the allocation of unnecessary program memory.

Listing 6.4 Big and small memory models

```
; Big memory model (3 16-bit words):  
move.l    address,d0  
  
; Small memory model (2 16-bit words):  
move.l    <address,d0
```

Certain instructions can be used only in small memory mode. If `<` is omitted in conjunction with these instructions, an error results. [Listing 6.5](#) shows the instruction `bmset.w`, which sets bit #zero in the specified address, and is valid only in small memory mode.

Listing 6.5 Small memory mode instruction

```
bmset.w #0001,<address
```

For maximum efficiency, it is recommended that you place data in the lower 64K of the memory map, in order to enable the compiler to use small memory mode.

Linker Command Files

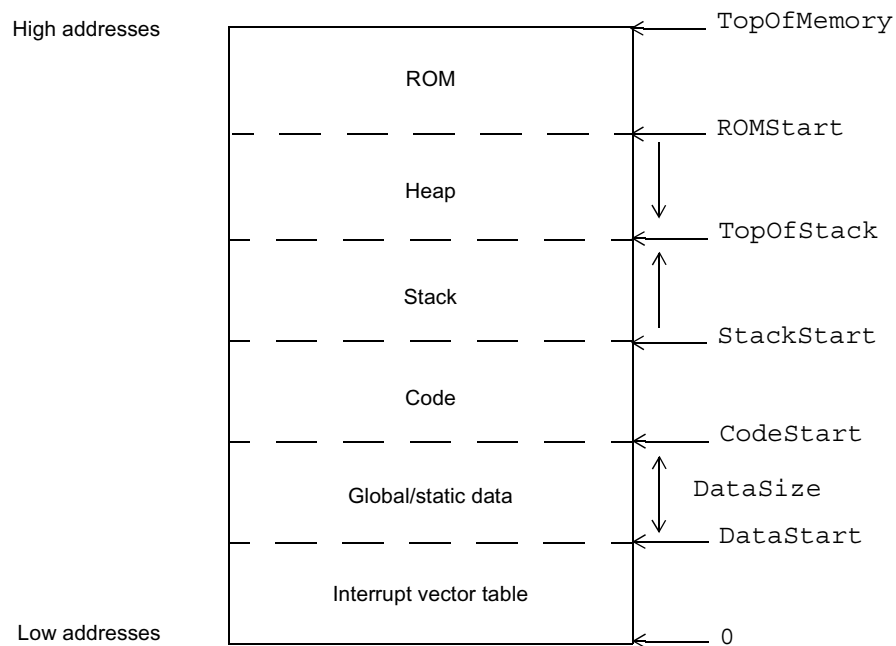
The SC100 Linker refers to a linker command file at link time, for various runtime values, addresses and labels. Two linker command files are provided, one for each memory mode. These files are `crtscsmm.cmd`, used in small memory mode, and `crtscbmm.cmd`, used when big memory mode is selected. Both files are located in the *install-dir/etc* directory.

Memory Layout and Configuration

The SC100 default memory layout is a single linear block which is divided into data and code areas. C programs generate code and data in sections. The compiler places each of these sections in its own continuous space in memory.

The default layout of the SC100 memory is illustrated in [Figure 6.1](#).

Figure 6.1 SC100 Default Memory Layout



Both memory models use the same default layout, but with different default values that define the distribution of the memory areas. You can change these default values. You also can configure the memory map to meet your specific requirements.

The layout and functionality of the stack and heap are common to both the small and big memory models.

The default memory map values for the small memory model are listed in [Table 6.2](#). These values are held in the file `crtscsmm.cmd`.

Table 6.2 Small Memory Model default values

From	Default value	To	Default value	Contents
0		0x1fff		Interrupt vector table
0x200		DataSize-1	0xffffd	Global and static variables
CodeStart	0x10000	StackStart-1	0x27fff	Program code
StackStart	0x28000	TopOfStack	0x7fff0	Stack and heap
ROMStart	0x80000	TopOfMemory	0xfffff	ROM

[Table 6.3](#) lists the default memory map values for the big memory model. These values are held in the file `crtscbmm.cmd`.

Table 6.3 Big Memory Model Default Values

From	Default value	To	Default value	Contents
0		0x1fff		Interrupt vector table
0x200		DataSize-1	0x17fff	Global and static variables
CodeStart	0x18000	StackStart-1	0x3ffff	Program code
StackStart	0x40000	TopOfStack	0x7fff0	Stack and heap
ROMStart	0x80000	TopOfMemory	0xfffff	ROM

Stack and Heap Configuration

The heap and stack are allocated from the same area of memory and must be contiguous. The compiler always treats the stack and heap as a continuous area of memory. The other sections of memory can be distributed, and there are no restrictions relating to their location.

Runtime stack

The compiler allocates an area of memory to the runtime stack, which is used for the following purposes:

- Allocation of local variables
- Passing arguments to functions
- Saving function return addresses
- Saving temporary results

The stack is allocated in the area above the space used for code, and grows in an upward direction toward the top of memory. The compiler uses the SP register to manage this stack.

The SC100 architecture includes two stack pointers:

- NSP, used when the processor is running in Normal mode
- ESP, used when the processor is running in Exception mode

As shown in [Table 6.1 on page 161](#), the default mode at initialization is Exception mode.

The compiler makes no assumptions about which stack pointer to use, and uses the pointer for the current processor mode to point to the address at the top of the stack.

When the system is initialized, the stack pointer for the current mode is set by default to the address of the location directly after the code area, as defined in `StackStart` in the linker command file. The actual address of the stack is determined at link time.

The stack pointer for the current processor mode is automatically incremented by the C environment at the entry to a function. This ensures that sufficient space is reserved for the execution of the function. At the function exit, the stack pointer is decremented, and the stack is restored to its previous size prior to function entry. If your application includes assembly language routines and C code, you must ensure at the end of each assembly routine that the current stack pointer is restored to its pre-routine entry state.

NOTE If you change the default memory configuration, remember to allow sufficient space for the stack to grow. If a stack overflow occurs at runtime, this will cause your program to fail. The compiler does not check for stack overflow during compilation or at runtime.

Dynamic memory allocation (heap)

The runtime libraries supported by the compiler include a number of functions which enable you to allocate memory dynamically for variables. See Chapter 7 for details of the runtime libraries supported. Since C does not support the dynamic allocation of memory, the compiler assigns an area of memory as a heap for this purpose.

The compiler allocates memory from a global pool for the stack and the heap together. The lower address of the area assigned to the stack and heap is defined in `StackStart`, in the linker command file. The heap starts at the top of memory, and is allocated in a downward direction toward the stack.

Objects that are dynamically allocated are addressed only with pointers, and not directly. The amount of space that can be allocated to the heap is limited by the amount of available memory in your system.

To make more efficient use of the space allocated to data, you can use the heap to allocate large arrays, instead of defining them as static or global.

For example, a definition such as `struct large array1[80];` can be defined using a pointer and the `malloc` function, as illustrated in [Listing 6.6](#).

Listing 6.6 Allocating large arrays from the heap

```
struct large *array1;  
array1 = (struct large *)malloc(80*sizeof(struct large));
```

Static Data Allocation

When you compile your application without cross-file optimization, the allocations for each file are assigned to different sections of data memory. At link time these are dispatched to different addresses.

When compiling with cross-file optimization, the compiler uses the same data section for all allocations. If you want to override this and to instruct the compiler to use non-contiguous data blocks, you can edit the machine configuration file to define the exact memory map of the system that you want to use.

Configuring the Memory Map

The default values in the SC100 memory map are easily configurable, by modifying the linker command file. When making such changes, it is important that you ensure that the code size and data size values that you specify do not overlap.

The stack and the heap must be always be located together in one contiguous area of memory. The compiler makes no assumptions about the layout of the other sections of memory, which can be split and distributed over non-contiguous parts of memory, as required.

NOTE If you choose not to modify the default linker command file, but rather save the changes in a new linker command file instead, use the `-mem` option to pass the new command file to the linker. If you use the `-Xlnk` option to do this, both the new linker command file and the default linker command file will be passed to the linker, resulting in errors.

Memory map configuration example

This example assumes that you have a system with non-contiguous memory, and would like to configure the memory as follows:

- All code placed in external memory (addresses 0x10000000 through 0x10100000)
- All data placed in internal memory
- Some local memory reserved for the most frequently used functions and overlays (addresses 0x10000 through 0x20000)
- All data placed in the lower 64K addresses, in order to be able to use the small memory model compilation mode

[Table 6.4](#) shows the memory map that meets the preceding list of requirements:

Table 6.4 Modified memory map configuration

From	To	Contents
0	0x1fff	Interrupt vectors
0x200	0xffffd	Global and static variables
0x10000	0x1ffff	Local code
0x20000	0x7fff0	Stack and heap
0x80000	0xfffff	ROM
0x10000000	0x100fffff	External code

[Listing 6.7](#) shows the definitions in the `crtscsmm.cmd` file that specify this memory map configuration:

Listing 6.7 Modified memory configuration in the linker command file

```
.provide _DataSize,    0x10000    ; Sets the data size.
.provide _CodeStart,  0x10000000  ; Sets the loader code
                                   ; start address.
.provide _StackStart, 0x20000     ; Sets the stack start
                                   ; address. The stack grows
                                   ; upward.
.provide _TopOfStack, 0x7fff0     ; The heap start address;
                                   ; the heap grows downward.
.provide _ROMStart,   0x80000     ; Sets the ROM start address.
```

Machine Configuration File

The machine configuration file contains the following:

- Information about data types and alignment requirements, used by the compiler for reference. This data must **not** be changed.
- Memory structure information, used by the compiler to allocate variables in the data sections of memory. This information can be modified if required.

By default, the compiler uses the file `proc.config`, located in the `install-dir/etc` directory. A different machine configuration file can be specified using the `-mc` option in the shell command line.

The SC100 memory structure consists of physical and logical memory maps, as follows:

- Physical memory is divided into several memory spaces. Each memory space is a physical entity consisting of a data bus and an address bus. A physical memory space is defined in terms of its size in words and the width of its address bus, and comprises blocks of words with contiguous addresses, described as physical memory areas.
- Logical memory areas are defined as blocks of memory words with contiguous addresses. These words are used by the compiler as if they were in physical memory areas. The addresses of the logical areas are mapped as offsets to physical memory addresses at link time.

This dual memory map structure provides a high degree of flexibility during the loading of application code.

Defining the memory configuration

Each memory space is defined individually in the machine configuration file, by specifying a space identifier and a description, comprising:

- Memory space type: program or data.
- Word size, in bytes.
- Area list, defining one or more logical areas.
- The addresses in the logical areas, as positive integers, used as offsets to physical memory areas.
- Physical area type: single-port RAM (ramsp), dual-port RAM (ramdp) or ROM (rom).
- Attached spaces (optional). This is used for dual-port RAM only, when ramdp is the defined area type, to specify the two memory spaces. It is important that the code ensures address consistency between the corresponding spaces.

The syntax for defining a memory space is as follows:

```
space definition:
    define space <space identifier>:
        space_type;
        word_size;
        area_list;
        end define
;
space_type:
    program | data
;
word_size:
    word : byte_number
;
area_list:
    area | area_list area
;
area:
    address_value .. address_value : area_type
opt_attached_spaces ;
;
area_type:
    ramsp | ramdp | rom
;
```

```
opt_attached_spaces:  
    [ space_number , space_number ] |  
;
```

In [Listing 6.8](#), a one-word data space is defined, providing one logical area that can be used for the allocation of variables.

Listing 6.8 Defining a data memory space

```
define space data_0 :  
    data;  
    word : 2;  
    0x0000 .. 0xffff : ramsp;  
end define
```

[Listing 6.9](#) shows the definition of a 2-word program space in ROM.

Listing 6.9 Defining a program memory space

```
define space pgm :  
    program;  
    word : 4;  
    0x0000 .. 0x3fff : rom;  
end define
```

At link time, these areas are mapped to the relevant physical memory space, and the actual addresses are calculated as offsets to the physical space starting address.

A data space can be divided into multiple logical areas, as shown in [Listing 6.10](#). When the compiler executes with cross-file optimization, it divides memory into these logical areas, and allocates variables accordingly.

Listing 6.10 Defining multiple memory spaces

```
define space data_1 :  
    data;  
    word : 2;  
    0x0000 .. 0x3fff : ramsp;  
    0x0800 .. 0xffff : ramdp [data_0,data_1];  
    0x10000 .. 0x13fff : ramsp;
```

```
    0x40000 .. 0x47fff : ramsp;  
end define
```

NOTE If you define new memory spaces in the machine configuration file, it is important that you also add these space definitions in the linker command file, to enable the linker to locate them at link time.

Application Configuration File

The application configuration file contains information about the interaction between the application software and the hardware. This file indicates to the compiler how to compile specific software units in order to ensure efficient sharing of hardware resources, in particular memory space. This information can be modified, to suit the requirements of your application.

The default application configuration file is named `minimal.appli`, and is located in the `install-dir/etc` directory. A different application configuration file can be specified, using the `-ma` option.

This file contains the following functional section types:

- Schedule section, which defines the entry points for the software units in the application and their overlay capabilities for local variables.
- Binding section, which specifies the links between software interrupt routines and hardware interrupt vectors, and between software-defined variables and fixed memory addresses.
- Overlay section, which specifies the overlay capabilities of global variables.

File structure and syntax

More than one section of each type can be included in the file. The order in which the sections are defined in the file is unimportant. Each of the section types is optional and can be omitted.

The syntax of the application configuration file is as follows:

```
translation_unit:  
    header_section  
    configuration section_list
```

```
    end configuration
;
header_section:
    opt_version
;
opt_version:
    version string_content |
;
section_list:
    section | section_list section
;
section:
    schedule_section | binding_section | overlay_section
;
```

Schedule section

The schedule section defines the entry point structure of an application, by specifying a “call tree”. The call tree root is a C function name that defines the starting entry point for an application. Each node in the call tree is the name of an entry point of a unit that can be called during the execution of the application.

Each call tree node is defined as a call tree item, and is given a `ct` number that is unique for the application. A call tree item can be one of three types:

- Background task, identifying the main entry point, defined as `main`
- Interrupt handler, identifying an interrupt routine entry point, defined as `it_entry`, with a number that is used by the binding section to link to the associated hardware interrupt vector
- Task entry point, defined as `task_entry`, for example, an operating system task

The schedule section can optionally include an overlay specification, which informs the compiler which groups of local variables can use the same memory location during execution of the application. The compiler is able to overlay groups of local variables automatically, but only when it is clear that the two sets of variables do not share the same lifetime, and are therefore not active simultaneously. By specifying overlays in this file, you provide the

necessary information in advance to help the compiler make more efficient use of memory space.

The overlay specification in the schedule section relates to local variables only. Overlays for global variables are specified in the overlay section.

The syntax of the schedule section is as follows:

```
schedule_list:
    schedule_elmt | schedule_list schedule_elmt
;
schedule_elmt:
    call_tree_list ; opt_overlay_spec
;
call_tree_list:
    call_tree_item | call_tree_list call_tree_item
;
call_tree_item:
    ct [int_constant] : main = ident ; |
    ct [int_constant] : it_entry int_constant = ident ; |
    ct [int_constant] : task_entry = ident ;
;
opt_overlay_spec:
    overlay = entry_overlay_list ; |
;
entry_overlay_list:
    [group_list]
;
group_list:
    group | group_list, group
;
group:
    [entry_number_list]
;
entry_number_list:
    entry_number | entry_number_list, entry_number
;
entry_number:
    ct[int_constant] | int_constant
;
```

[Listing 6.11](#) defines two entry points, in addition to `main`. The function `task1()` is defined as a task entry point and the function `int_entry()` is defined as an interrupt handler.

NOTE Defining a function as an interrupt handler in the application configuration file is equivalent to using `#pragma interrupt` in the source file.

Listing 6.11 Defining additional entry points for an application

```
configuration

schedule
    ct[0] : main = _main;
    ct[1] : task_entry = _task1;
    ct[2] : it_entry 0 = _int_entry;
end schedule

binding
    place ___stackX on space 0 at 1;
end binding

end configuration
```

Binding section

The binding section performs the following functions:

- Assignment of fixed memory addresses to variables. A full memory address is specified with a memory binding directive, using the following syntax:

```
memory_binding_directive:
place full_ident on space_identifier at number
```

- Specification of the links between fixed interrupt entries and hardware interrupt vector addresses. An interrupt binding directive is used to specify an interrupt entry number, in the range 0-15, and the corresponding hardware vector number, in the range 1-16, using the following syntax:

```
it_binding_directive:  
place it_vector interrupt_number on space_identifier at  
vector_number
```

The syntax of the binding section is as follows:

```
binding_directive:  
    memory_binding_directive | it_binding_directive  
;  
binding_directive_list:  
    binding_directive binding_directive_list ; binding_directive  
;  
binding section:  
    binding  
        binding_directive_list  
    end binding  
;  
;
```

In [Listing 6.12](#), the location of global variable `mem` is fixed at absolute address `0x2000`:

Listing 6.12 Placing a variable at an absolute location

```
configuration  
  
schedule  
    ct[0] : main = _main;  
    ct[1] : it_entry 0 = _int_entry;  
end schedule  
  
binding  
    place __stackX on space 0 at 1;  
    place _mem on space 0 at 0x2000;  
end binding  
  
end configuration
```

Overlay section

The overlay section specifies how the compiler should overlay global variables in order to further reduce the amount of memory

required for data. As with local variables, in many cases the compiler can automatically detect that two data objects do not share the same lifetime and as a result, the memory allocated to these objects can be shared. This feature is needed for cases where the compiler cannot identify statically that the object lifetimes of global variables do not conflict.

Defining the overlay specification for global variables includes the following:

- Grouping the global variables into sets that can share the same memory space. In the overlay section syntax, the full identity is specified for each global variable, or list of variables, and defined as `symbol_list`.
- Defining each set of global variables as a `symbol_group`, associated with a `symbol_list` and an identifying group number.
- Specifying compatibility clauses that define which symbol groups can be overlaid, using the keyword `discern`.
- Specifying a list of compatibility clauses to indicate which symbol groups in the application can share the same memory space.

The syntax of the overlay section is as follows:

```
overlay section:
    overlay
        opt_overlay_spec
        compatibility_list
    end overlay
;
symbol_list:
    full_ident | symbol_list, full_ident
;
symbol_group:
    SG [number] = [symbol_list] ;
;
symbol_group_list:
    symbol_group | symbol_group_list symbol_group
;
sg_ref:
    SG [number]
;
sg_list:
```

```
    sg_ref | sg_list, sg_ref
;
compatibility_clause:
    discern_sg_ref : sg_list ;
;
compatibility_list:
    compatibility_clause | compatibility_list
compatibility_clause
;

```

[Listing 6.13](#) shows an overlay section that specifies that the application will never access the two global arrays, `arr1` and `arr2`, at the same time, and they can therefore share the same physical memory location.

Listing 6.13 Defining global variable overlays

```
configuration
schedule
    ct[0] : main = _main;
end schedule

binding
    place ___stackX on space 0 at 1;
end binding

overlay
    sg[0] = [_arr1];
    sg[1] = [_arr2];
    discern sg[0] : sg[1];
end overlay
end configuration

```

Calling Conventions

The compiler supports a stack-based calling convention. Additional calling conventions are also supported. Calling conventions can be mixed within the same application.

Specific calling conventions can be enforced using pragmas.

When compiling in separate compilation mode, non-static functions use the stack-based calling convention.

Stack Pointer

The SP register serves as the stack pointer, which points to the first available location. The stack direction is toward higher addresses, meaning that a push is implemented as `(sp) +`. The stack pointer must always be 8-byte aligned.

Stack-Based Calling Convention

The following calling conventions are supported:

- The first (left-most) function parameter is passed in `d0` if it is a numeric scalar or in `r0` if it is an address parameter, regardless of its size. The second function parameter is passed in `d1` if it is a numeric scalar, or in `r1` if it is an address parameter, regardless of its size. The remaining parameters are pushed on the stack. Long parameters are pushed on the stack using little-endian mode, with the least significant bits in the lower addresses.
- Structures and union objects that can fit in a register are treated as numeric parameters, and are therefore candidates to be passed in a register.
- Numeric return values are returned in `d0`. Numeric address return values are returned in `r0`. Functions returning large structures, meaning structures that do not fit in a single register, receive and return the returned structure address in `r2`. The space for the returned object is allocated by the caller.
- Functions with a variable number of parameters allocate all parameters on the stack.
- Parameters are aligned in memory according to the base parameter type, with the exception of characters and unsigned characters that have a 32-bit alignment.

The following registers are saved by the caller: d0-d5, r0-r5, n0-n3.

The following registers are saved by the callee, if actually used: d6-d7, r6-r7.

The compiler assumes that the current settings of the following operating control bits are correct:

- Saturation mode
- Round mode
- Scale bits

The application is responsible for setting these mode bits correctly.

[Listing 6.14](#) shows two function calls and the parameters that are allocated for each call.

Listing 6.14 Function call and allocation of parameters

```
# Function call:
foo(int a1, struct fourbytes a2, struct eightbytes a3, int *a4)

# Parameters for the preceding function call:
# a1 - in d0
# a2 - in d1
# a3 - in stack
# a4 - in stack

# Function call:
bar(long *b1, int b2, int b3[])

# Parameters for the preceding function call:
# b1 - in r0
# b2 - in d1
# b3 - in stack
```

The stack-based calling convention must be used when calling functions that are required to maintain a calling stack.

The compiler is able to use optimized calling sequences for functions that are not exposed to external calls.

Local and formal variables are allocated on the stack and in registers.

[Table 6.5](#) summarizes register usage in the stack-based calling convention.

Table 6.5 Register usage in the stack-based calling convention

Register	Used as	Caller Saved	Callee Saved
d0	First numeric parameter Return numeric value	+	
d1	Second numeric parameter	+	
d2-d5		+	
d6-d7			+
d8-d15		+	
r0	First address parameter Return address value	+	
r1	Second address parameter	+	
r2	Big object return address	+	
r3-r5		+	
r6	Optional argument pointer		+
r7	Optional frame pointer		+
n0-n3, m0-m3		+	

Optimized Calling Sequences

A stack-less convention may be used when calling functions that are not reentrant, if this technique generates more efficient code than other conventions.

This convention will be used only if the function is not visible to external code.

When using this calling convention, local variables may be allocated statically, meaning not on a stack. Functions with mutually exclusive lifetimes may share space for their local variables.

Actual parameters are placed by the calling function at the locations allocated for the formal parameters in the called function. The compiler may use registers and memory locations as required when allocating locations for the formal parameters.

Under this calling convention, all registers are classified as caller-saved.

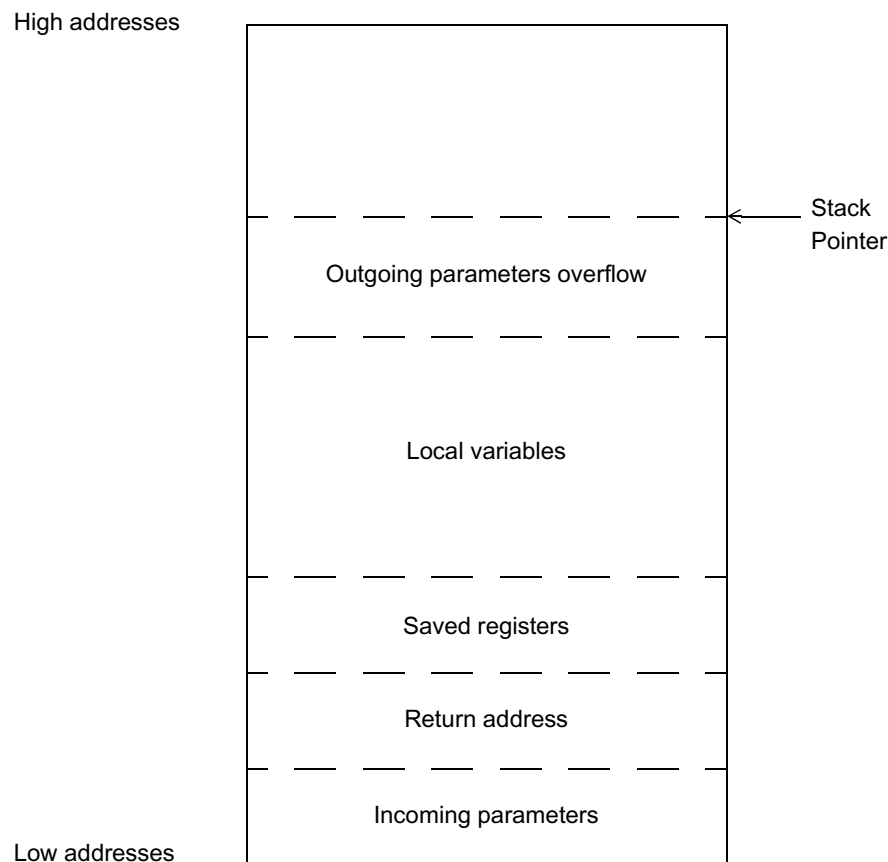
Return values from functions are placed in the space allocated for the function return value in the calling function. The compiler may use a register or a memory location as the space for the function return value.

Stack Frame Layout

The stack pointer points to the top (high address) of the stack frame. Space at higher addresses than the stack pointer is considered invalid and may actually be unaddressable. The stack pointer value must always be a multiple of eight.

[Figure 6.2](#) shows typical stack frames for a function, indicating the relative position of local variables, parameters and return addresses.

Figure 6.2 **Stack Frame Layout**



The caller must reserve stack space for return variables that do not fit in registers. This return buffer area is typically located with the local variables. This space is typically allocated only for functions that make calls that return structures. Beyond these requirements, a function is free to manage its stack frame as necessary.

The outgoing parameter overflow block is located at the top (higher addresses) of the frame. Any incoming argument spill generated for `varargs` and `stdargs` processing must be at the bottom (low addresses) of the frame.

The caller puts argument variables that do not fit in registers into the outgoing parameter overflow area. If all arguments fit in registers, this area is not required. A caller has the option to allocate argument overflow space sufficient for the worst case call, use portions of this space as necessary, and/or leave the stack pointer unchanged between calls.

Local variables that do not fit into the local registers are allocated space in the local variables area of the stack. If there are no such variables, this area is not required.

Creating a Calling Convention

The default setting for the compiler is to use the calling convention defined in the ABI document. However, there are situations where the default setting is not the best choice.

One common situation arises when the compiler must call an assembly function. If the compiler calls the assembly function in only one module, it is possible to use the `asm` prefix on a function in combination with the `asm arg` descriptor mechanism. The body of this function contains the assembly program. The issue is that the `asm` prefix is exclusive with `extern` and therefore you cannot export this function.

To avoid this limitation, you can define a specific calling convention in the application configuration file and then instruct the compiler to use this specific calling convention for a given function using a `pragma`.

You must describe user-defined calling conventions have to described in the application configuration file. (You pass this file to the compiler using the `-ma file_name` option.)

[Listing 6.15](#) shows the calling convention syntax.

Listing 6.15 Calling convention syntax

```
<A call convention> ::=
    call_convention <Name> (
        arg [ <Arg descriptor> ] ;
        return <One Reg> ;
        saved_reg [ <Register list> ] ;
        deleted_reg [ <RegisterList> ] ;
        <save_protocol>)

<Arg descriptor> ::= <One Arg> | <Arg descriptor> , <One Arg>

<One Arg> ::= <arg number> : <One Reg>

<arg number> ::= <A positive integer>

<One Reg> ::= <One Phi Reg> |
    <One Phi Reg> fract /* This register is treated as a
                        fractional, that is, left aligned. */
    |
    ( * <One Phi Reg> , <One Phi Reg> ) /* * Means that the argument
                                        is of pointer type */

<Register list> ::= <One Reg> | <Register list> , <One Reg>

<One Phi Reg> :=
    $d0 | $d1 | $d2 | $d3 | $d4 | $d5 | $d6 | $d7 | $d8 | $d9 | $d10
    | $d11 | $d12 | $d13 | $d14 | $d15 | $r0 | $r1 | $r2 | $r3 | $r4
    | $r5 | $r6 | $r7 | $r8 | $r9 | $r10 | $r11 | $r12 | $r13 | $r14
    | $r15 | $m0 | $m1 | $m2 | $m3 | $b0 | $n1 | $n2 | $n3

<save_protocol> ::=
    save = [ ] ; /* protocol is push and pop */
    |
    save = [ <save_rt_name> , <restore_rt_name> , <frame_effect> ] ;
```

```
/* Protocol is save and restore through user defined runtime.
   The restore run time is not expected to do the return of
   the callee. frame_effect is the number of bytes consumed
   on the stack */
|
save = [ return , <save_rt_name> , <restore_rt_name> ,
<frame_effect> ] ; /* Protocol is save and restore through
                    user defined runtime. The restore run
                    time will also take care of the return
                    of the callee. In this case, the compiler
                    remove the return in the caller.
                    frame_effect is the amount of byte
                    consumed on the stack */
|
/* Default is push and pop */
;
```

You can create several calling conventions as long as they use different names.

Argument descriptor section

This section defines how the input arguments are passed.

General mechanism

```
<One Arg> ::=
    <arg number> : <One Reg> ;
```

Arguments are numbered from 1 to n (not necessarily by increments of one); this is the `<arg_number>`.

Argument number refers to the position in the argument list of the function on which the calling convention has to be applied.

If the C function using this calling convention has more arguments than defined in the calling convention, the default rule applies; that is, unspecified arguments are passed on the stack. The same holds for any function argument not defined in the calling convention: it goes on the stack.

`<One Reg>` is the name of a register to be used to passed the parameter.

Function returning a structure

The compiler automatically translates the code shown in [Listing 6.16](#) into the code shown in [Listing 6.17](#).

Listing 6.16 Function Returning a Structure

```
typedef struct {  
    int a;  
    int b;  
} My_Type;  
  
My_Type A_Function()  
{  
}
```

Listing 6.17 Translated Code for Function Returning a Structure

```
void A_Function(My_Type *Compiler_Internal_Name)
```

(In the preceding special case, the argument was numbered as argument zero.)

Return descriptor

```
return <One Reg> ;
```

The return descriptor provides the name of the register in which the return value is returned. Functions are always expected to pass the return value in a register. Therefore, this section is required. (Keep in mind the special case of functions returning structures.)

Saved register section

```
saved_reg [ <Register list> ] ;
```

This section defines the list of registers that the function must save if they are used inside the function. The effect is for the caller to be able to safely keep a value in one of these registers during a call to a function obeying this calling convention.

Deleted register section

```
deleted_reg [ <RegisterList> ] ;
```

This function defines the list of registers deleted, or potentially deleted, by this function. The effect is that the caller cannot keep a value in one of these registers during a call to a function obeying this calling convention.

General remarks

Loop-related registers are not part of the registers. This is due to the fact that the save/restore of loop-related registers is expensive on this machine. Therefore, the compiler automatically disables hardware loop and modulo transformation in a loop when this loop contains calls.

The compiler also assumes that when a function is left it is left in a consistent state, that is, no hardware loops are active and all pointers are set to linear mode. Again, the compiler enforces this for C routines but when the called routine is an assembly routine this is the responsibility of the user.

To specify the calling convention for a function, use the `call_conv` pragma.

To specify the calling convention for an entire file, use the `default_call_conv` pragma.

User-Defined Calling Convention Examples

This section shows some user-defined calling convention examples.

Example 1

[Listing 6.18](#) shows an example application configuration file named `call.appli` that contains two user-defined calling conventions.

Listing 6.18 Example 1: user-defined calling conventions

```
/* This is a user defined calling convention defined as:
Argument 1 if it exists is passed in $d0
Argument 2 if it exists is passed in $d7
Argument 3 if it exists is passed in $r2
If other arguments exist they are passed on the stack

Return value (if it exists) will be in register $d10
the saved_reg list is going to be pushed/poped (or are unmodified)
by the callee the deleted_reg list is the set of registers whose
content is not valid after the call*/
```

configuration

```
call_convention Call_Conv_1 (  
    arg [1 : $d0, 2 : $d7, 3 : $r2];  
    return $d10;  
    saved_reg [$d1, $d2, $r6, $r7];  
    deleted_reg [$d0, $d3, $d4, $d5, $d6, $d7, $d8, $d9, $d10,  
                $d11, $d12, $d13, $d14, $d15, $r0, $r1, $r2, $r3,  
                $r4, $r5, $r8, $r9, $r10, $r11, $r12, $r13, $r14,  
                $r15];  
    save = [ ];  
)  
  
call_convention Call_Conv_2 (  
    arg [1 : $d0, 2 : $d1, 3 : $d3, 4 : $d4, 5 : $d5, 6 : $d6,  
        7 : $d7, 8 : $d8];  
    return $d0;  
    saved_reg [$d2, $d3, $d4, $d5, $d6, $d7, $d8, $d9,  
                $d10, $d11];  
    deleted_reg [$d0, $d1, $d12, $d13, $d14, $d15, $r0, $r1,  
                $r2, $r3, $r4, $r5, $r6, $r7, $r8, $r9, $r10,  
                $r11, $r12, $r13, $r14, $r15];  
    save = [ ];  
)  
  
end configuration
```

[Listing 6.19](#) shows `call.c`, which shows the following setup for calling conventions:

- The compiler calls the `fct` function using `Call_Conv_1`.
- The compiler calls the `fct1` function using `Call_Conv_2`.
- The compiler calls the `main` function using the ABI default calling convention.

Listing 6.19 Example 1: code that uses user-defined calling conventions

```
#include <stdio.h>  
volatile int val = 10;
```

Runtime Environment

Creating a Calling Convention

```
int fct(int a, int b, int c, int d,int e,int f,int g,int h)
{
#pragma noline
#pragma call_conv fct Call_Conv_1
    return (a + b + c + d + e + f + g + h);
}

int fct1(int a, int b, int c, int d,int e,int f,int g,int h)
{
#pragma noline
#pragma call_conv fct1 Call_Conv_2
    int i;
    int acc = 0;

    for (i =0; i<val; i++) {
        acc += fct(a, b, c, d,e,f,g,h);
        a++;b++;c++;d++;e++;f++;g++;h++;
    }

    return (acc + a + b + c + d + e + f + g + h);
}

void main()
{
    int i;
    int a, b, c, d,e,f,g,h;
    int Z = 0;

    a=b=c=d=e=f=g=h=0;

    for (i =0; i<val; i++) {
        Z += fct1(a, b, c, d,e,f,g,h);
        a++;b++;c++;d++;e++;f++;g++;h++;
    }

    Z = Z + a + b + c + d + e + f + g + h;

    printf("Z = %d\n", Z);
}
```

To compile the preceding test case, type the following commands:

```
scc -ma call.appli call.c
```

Example 2

You also can use user-defined calling conventions with assembly language functions.

Instead of using the usual approach where everything has to be defined after the `asm_header` keyword, (that is, in which register arguments are passed, set of deleted registers, and so on) you can use a single directive:

```
.call_conv = <call_conv_name>
```

Using the single `.call_conv` directive has these benefits:

- You can use a more compact description.
- The possibility exists to call a C assembly function from outside the current module without having to use the ABI rules for calling conventions. You place a `pragma call_conv` on the C assembly function when declaring it as an `extern` in the module that uses it.

[Listing 6.20](#) shows a user-defined calling convention that is convenient for using with assembly functions.

Listing 6.20 Example 2: user-defined calling conventions for assembly functions (call.appli)

```
configuration
  call_convention Call_Conv_1 (
    arg [1 : (* $d14, $d14), 2 : (* $r9, $r9)];
    return (* $d0, $d0);
    deleted_reg [$d0, $r9, $d14];
    save = [ ];
  )
end configuration
```

[Listing 6.21](#) shows the use of the `.call_conv` directive with user-defined calling conventions and assembly functions.

Listing 6.21 Example 2: using `.call_conv` directive with assembly functions (File1.c)

```
asm int My_Asm_Function(int a, int *b)
{
```

```
asm_header
.call_conv = "Call_Conv_1";
/* call_conv replaces the commented section; in this case we
   do not follow the ABI rules for register passing
   .arg
       _a in $d14;
       _b in $r9;
   return in $d0;
   .reg $d0, $r9, $d14;
*/

asm_body
    move.l d14,d0
    move.l (r9),d14
    add d14,d0,d0
asm_end
}
```

[Listing 6.22](#) shows the use of the call_conv pragma with user-defined calling conventions and assembly functions.

Listing 6.22 Example 2: using call_conv pragma with assembly functions (File2.c)

```
#include <stdio.h>

extern int My_Asm_Function(int a, int *b);
#pragma call_conv My_Asm_Function Call_Conv_1

int Buff[10] = {25};
int Ret;

void main()
{
    Ret = My_Asm_Function(10, Buff);
    /* 10 is going to be passed in d14, Buff in r9, and Ret
       will be in d0 */
    printf("Ret = %d\n", Ret);
}
```

To compile and run the preceding example, type the following commands:

```
scc -ma call.appli File1.c File2.c  
runsc100 a.eld
```

The correct result of the example is 35.

Interrupt Handlers

Functions which require no parameters and return no result can be designated as interrupt handler functions. The process of creating an interrupt handler function includes:

- Defining the function as an interrupt handler
- Linking the function to the appropriate interrupt vector entry

An interrupt handler can be defined in one of two ways:

- Using `#pragma interrupt` in the source code
- Defining an interrupt entry point in the application, by editing the schedule section of the application configuration file

To create the link between the function and the interrupt vector entry, you can use any one of the following options:

- In the code that calls the function, place a call to the handler function in the interrupt vector entry.
- Use the `signal.h` library function to insert a call to the interrupt handler function into the required interrupt vector entry.
- If the function is very small, you can embed it in the interrupt vector entry, by modifying the startup code file, `crtsc100.asm`. The size of each interrupt vector entry is 64 bytes. With this option, there is no need for an explicit call from the vector to the function.

Interrupt handler functions always follow the stack-based calling convention. When an interrupt function is called, the interrupt handler saves all registers and all other resources that are modified by the function. Upon returning from the function all registers and hardware loop state saved at entry are restored to their original state.

Local variables are saved on the stack. Interrupt handlers that are known to be non-interruptible may also allocate data statically.

Return from interrupt is implemented using an RTE instruction.

Frame Pointer and Argument Pointer

The compiler does not use a frame pointer or an argument pointer.

If, however, the use of a frame pointer or an argument pointer is required by external code, `r7` may be allocated as a frame pointer and `r6` as an argument pointer. When these registers are allocated as frame pointer and/or argument pointer they should be saved and restored as part of the function prolog/epilog code.

Hardware Loops

All hardware loop resources are available for use by the compiler. It is assumed that no nesting occurs when entering a function. As a result, a function may use all 4 nesting levels for its own use. An additional side effect of this assumption is that loops that include a function call as part of the loop code cannot be implemented using hardware loops, unless the compiler can infer the nesting level of the called function from static variables known at compilation time.

Loops are nested beginning with loop counter 3 at the innermost nesting level.

Operating Modes

The compiler makes the following assumptions regarding runtime operating modes and the machine state:

- All modulo (M) registers (`m0-m3`) are assumed to contain the value -1 (linear addressing). If the use of an M register is required, the using function must restore the M register to the value -1 before returning or before calling another function.
- No specific settings are assumed for the operating mode settings in the OMR register. The compiler assumes that the default settings in the startup code, including saturation modes, rounding mode and scale bits, are set by the user. You can control and change these operating modes during execution of the application. Refer to the SC100 architecture documentation for further details.

Runtime Libraries

This chapter describes the C libraries and I/O libraries supported by the Metrowerks Enterprise C compiler. Each table in this chapter is organized in alphabetical order, according to the file, function, or constant name in the first column in the table.

[Table 7.1](#) summarizes the ISO standard C libraries that the compiler supports.

Table 7.1 Supported ISO libraries

Header file	Description
<code>ctype.h</code>	Character typing and conversion
<code>float.h</code>	Floating-point characteristics
<code>limits.h</code>	Integer characteristics
<code>locale.h</code>	Locales
<code>math.h</code>	Floating-point math
<code>setjmp.h</code>	Nonlocal jumps
<code>signal.h</code>	Signal handling
<code>stdarg.h</code>	Variable arguments
<code>stddef.h</code>	Standard definitions
<code>stdio.h</code>	I/O library
<code>stdlib.h</code>	General utilities
<code>string.h</code>	String functions
<code>time.h</code>	Time functions

The non-ISO C libraries supported by the compiler are shown in [Table 7.2](#). This library contains the built-in intrinsic functions supplied with the compiler. Which header file you use to include the library depends on whether your code may have a conflict between certain assembly language operations and intrinsic functions.

Table 7.2 Supported non-ISO libraries

Header file	Description
<code>prototype.h</code>	Built-in intrinsic functions.
<code>prototype_asm.h</code>	An alternative header file that defines built-in intrinsic functions so that they do not conflict with the following assembler operations: <code>add</code> , <code>debug</code> , <code>debugv</code> , <code>di</code> , <code>ei</code> , <code>max</code> , <code>mark</code> , <code>min</code> , <code>mpyuu</code> , <code>mpysu</code> , <code>mpyus</code> , <code>stop</code> , <code>sub</code> , <code>trap</code> , <code>wait</code>
	If needed, include this file in your code instead of <code>prototype.h</code> .

Character Typing and Conversion (*ctype.h*)

The `ctype.h` library contains the following function types:

- Testing functions
- Conversion functions

Testing Functions

[Table 7.3](#) lists the testing functions that the compiler supports.

Table 7.3 Testing functions

Function	Purpose
<code>int isalnum(int)</code>	Tests for <code>isalpha</code> or <code>isdigit</code>
<code>int isalpha(int)</code>	Tests for <code>isupper</code> or <code>islower</code>
<code>int iscntrl(int)</code>	Tests for any control character
<code>int isdigit(int)</code>	Tests for decimal digit character
<code>int isgraph(int)</code>	Tests for any printing character except space
<code>int islower(int)</code>	Tests for lowercase alphabetic character
<code>int isprint(int)</code>	Tests for any printing character including space
<code>int ispunct(int)</code>	Tests for any printing character not space and not <code>isalnum</code>

Table 7.3 Testing functions

Function	Purpose
<code>int isspace(int)</code>	Tests for white-space characters
<code>int isupper(int)</code>	Tests for uppercase alphabetic character
<code>int isxdigit(int)</code>	Tests for hexadecimal digit character

Conversion Functions

[Table 7.4](#) lists the conversion functions that the compiler supports.

Table 7.4 Conversion functions

Function	Purpose
<code>int tolower(int)</code>	Converts uppercase alphabetic character to the equivalent lower case character
<code>int toupper(int)</code>	Converts lowercase alphabetic character to the equivalent uppercase character

Floating Point Characteristics (float.h)

The compiler represents floating point numbers using IEEE format (ANSI/IEEE Std 754-1985). Only single precision floating point format is supported.

The contents of `float.h` are listed in [Table 7.5](#).

Table 7.5 Contents of file float.h

Constant	Value	Purpose
<code>FLT_DIG</code> <code>DBL_DIG</code> <code>LDBL_DIG</code>	6 6 6	Number of decimal digits of precision
<code>FLT_EPSILON</code> <code>DBL_EPSILON</code> <code>LDBL_EPSILON</code>	1.1920929E-07 1.1920929E-07 1.1920929E-07	Minimum positive number χ such that $1.0 + \chi$ does not equal 1.0
<code>FLT_MANT_DIG</code> <code>DBL_MANT_DIG</code> <code>LDBL_MANT_DIG</code>	24 24 24	Number of base-2 digits in the mantissa

Table 7.5 Contents of file float.h

Constant	Value	Purpose
FLT_MAX_10_EXP	38	Maximum positive integers n such that 10^n is representable
DBL_MAX_10_EXP	38	
LDBL_MAX_10_EXP	38	
FLT_MAX_EXP	128	Maximum positive integer n such that 2^{n-1} is representable
DBL_MAX_EXP	128	
LDBL_MAX_EXP	128	
FLT_MAX	3.4028235E+38	Maximum positive floating point number
DBL_MAX	3.4028235E+38	
LDBL_MAX	3.4028235E+38	
FLT_MIN_10_EXP	-39	Minimum negative integer n such that 10^n is representable
DBL_MIN_10_EXP	-39	
LDBL_MIN_10_EXP	-39	
FLT_MIN_EXP	-126	Minimum negative integer n such that 2^{n-1} is representable
DBL_MIN_EXP	-126	
LDBL_MIN_EXP	-126	
FLT_MIN	5.8774817E-39	Minimum positive number
DBL_MIN	5.8774817E-39	
LDBL_MIN	5.8774817E-39	
FLT_RADIX	2	Floating point exponent is expressed n radix 2.
FLT_ROUNDS	1	
		Floating point rounding is to nearest even number.

Floating Point Library Interface (fltmath.h)

This header file defines the software floating point library interface. Most of these functions are called by the code generator of the compiler for floating point expression evaluation. They may also be called directly by user code.

The floating point library supports the full IEEE-754 single-precision floating point standard.

Three configuration parameters and one status word can be used. Each of these is described in the following sections.

- Round_Mode
- FLUSH_TO_ZERO
- IEEE_Exceptions
- EnableFPExceptions

Round_Mode

Four rounding modes are supported:

- `ROUND_TO_NEAREST_EVEN`. The representable value nearest to the infinitely precise intermediate value is the result. If the two nearest representable values are equally near (tie), then the one with the least significant bit equal to zero (even) is the result.
- `ROUND_TOWARDS_ZERO`. The result is the value closest to, and no greater in magnitude than, the infinitely precise intermediate result.
- `ROUND_TOWARDS_MINUS_INF`. The result is the value closest to and no greater than the infinitely precise intermediate result (possibly minus infinity).
- `ROUND_TOWARDS_PLUS_INF`. The result is the value closest to and no less than the infinitely precise intermediate result (possibly plus infinity).

By default, the rounding mode is set to `ROUND_TO_NEAREST_EVEN`.

[Listing 7.1](#) shows an example of changing the round mode to `ROUND_TOWARDS_MINUS_INF`.

Listing 7.1 Changing the round mode

```
#include <fltmath.h>
. . .
Round_Mode = ROUND_TOWARDS_MINUS_INF.
```

FLUSH_TO_ZERO

This is a boolean configuration item that sets the behavior of un-normalized numbers. When set to true (default) all un-normalized values are flushed to zero. This leads to better performance, but a smaller dynamic range.

For example, [Listing 7.2](#) shows how to disable the `FLUSH_TO_ZERO` option.

Listing 7.2 Disabling flushing to zero

```
#include <fltmath.h>
. . .
FLUSH_TO_ZERO = 0;
```

IEEE_Exceptions

This is a status word that represents the IEEE exceptions that were raised during the last floating point operation. By default, the floating point library sets these values but does not handle any of these exceptions.

The following exceptions are supported:

- IEEE_Inexact
- IEEE_Divide_By_Zero
- IEEE_Underflow
- IEEE_Overflow
- IEEE_Signaling_Nan

See the IEEE standard for the exact description of these exceptions.

[Listing 7.3](#) shows an example of how to use the exception status word.

Listing 7.3 Using the exception status word

```
#include <fltmath.h>
float x,y;
. . .
x = x*y;
if (IEEE_Exceptions & IEEE_Overflow)
{
<handle overflow>
}
```

EnableFPExceptions

This is a bit field mask. Setting a flag enables raising an SIGFPE signal if the last FP operation raised this exception.

For example, [Listing 7.4](#) shows an example that installs a signal for handling overflow and divide by zero exceptions.

Listing 7.4 Setting a signal for exceptions

```
#include <fltmath.h>
#include <signal.h>
void SigFPHandler(int x)
```

```
{
switch (IEEE_Exceptions)
{
case IEEE_Overflow:
. . .
case IEEE_Divide_by_zero:
. . .
}
}
float x,y;
. . .
EnableFPEExceptions = IEEE_Overflow | IEEE_Divide_by_zero;
signal(SIGFPE, SigFPHandler)
x = x*y    /*This will raise SIGFPE if overflow or */
          divide by zero occur */
```

NOTE Because the signal handling installs the handler address into the interrupt table, this example works only if the interrupt vector table is located in RAM. If the call to `SIGNAL` is not able to install the new handler, `SIG_ERR` is returned.

Integer Characteristics (limits.h)

The contents of `limits.h` are listed in [Table 7.6](#).

Table 7.6 Contents of file limits.h

Constant	Value	Purpose
CHAR_BIT	8	Width of char type, in bits
CHAR_MAX	127	Maximum value for char
CHAR_MIN	-128	Minimum value for char
INT_MAX	2147483647	Maximum value for int
INT_MIN	(-2147483647-1)	Minimum value for int
UINT_MAX	429496729u	Maximum value for unsigned int
LONG_MAX	2147483647	Maximum value for long int
LONG_MIN	(-2147483647-1)	Minimum value for long int
ULONG_MAX	429496729uL	Maximum value for unsigned long int
MB_LEN_MAX	2	Maximum number of bytes in a multibyte character

Table 7.6 Contents of file limits.h

Constant	Value	Purpose
SCHAR_MAX	127	Maximum value for signed char
SCHAR_MIN	-128	Minimum value for signed char
UCHAR_MAX	255	Maximum value for unsigned char
SHRT_MAX	32767	Maximum value for short int
SHRT_MIN	-32768	Minimum value for short int
USHRT_MAX	65536u	Maximum value for unsigned short int

Locales (locale.h)

[Table 7.7](#) lists the locales functions that the compiler supports.

Table 7.7 Locale functions

Function	Purpose
localeconv(void)	Not applicable
setlocale(int category, const char* locale)	Not applicable

NOTE The functions listed in [Table 7.7](#) are supported for compatibility purposes and have no effect.

Floating Point Math (math.h)

The `math.h` library contains the following function types:

- Trigonometric functions
- Hyperbolic functions
- Exponential and logarithmic functions
- Power functions
- Other functions

The compiler runtime environment fully implements the `math.h` library using floating point emulation.

Trigonometric Functions

[Table 7.8](#) lists the trigonometric functions that the compiler supports.

Table 7.8 Trigonometric functions

Function	Purpose
<code>double acos(double)</code>	arc cosine
<code>double asin(double)</code>	arc sine
<code>double atan(double)</code>	arc tangent
<code>double atan2(double, double)</code>	arc tangent2
<code>double cos(double)</code>	cosine
<code>double sin(double)</code>	sine
<code>double tan(double)</code>	tangent

Hyperbolic Functions

[Table 7.9](#) lists the hyperbolic functions that the compiler supports.

Table 7.9 Hyperbolic functions

Function	Purpose
<code>double cosh(double)</code>	Hyperbolic cosine
<code>double sinh(double)</code>	Hyperbolic sine
<code>double tanh(double)</code>	Hyperbolic tangent

Exponential and Logarithmic Functions

[Table 7.10](#) lists the exponential and logarithmic functions that the compiler supports.

Table 7.10 Exponential and logarithmic functions

Function	Purpose
<code>double exp(double)</code>	Exponential
<code>double frexp(double, int*)</code>	Splits floating point into fraction and exponent
<code>double ldexp(double, int)</code>	Computes value raised to a power
<code>double log(double)</code>	Natural logarithm
<code>double log10(double)</code>	Base ten (10) logarithm
<code>double modf(double, double*)</code>	Splits floating point into fraction and integer

Power Functions

[Table 7.11](#) lists the power functions that the compiler supports.

Table 7.11 Power functions

Function	Purpose
<code>double pow(double, double)</code>	Raises value to a power
<code>double sqrt(double)</code>	Square root

Other Functions

[Table 7.12](#) lists the other functions that the compiler supports.

Table 7.12 Other functions

Function	Purpose
<code>double ceil(double)</code>	Ceiling
<code>double fabs(double)</code>	Floating point absolute number
<code>double floor(double)</code>	Floor
<code>double fmod(double, double)</code>	Floating point remainder

Nonlocal Jumps (setjmp.h)

[Table 7.13](#) lists the nonlocal jumps that the compiler supports.

Table 7.13 Nonlocal jumps

Function	Purpose
<code>typedef unsigned int jmp_buf[32]</code>	Buffer used to save the execution context
<code>void longjmp(jmp_buf, int)</code>	Nonlocal jump
<code>int setjmp(jmp_buf)</code>	Nonlocal return

Signal Handling (signal.h)

[Table 7.14](#) lists the signal handling that the compiler supports.

Table 7.14 Signal handling (signal.h)

Function	Purpose
<code>int raise(int)</code>	Raises a signal
<code>void(*signal(int, void (*)(int)))(int)</code>	Installs a signal handler

Variable Arguments (stdarg.h)

[Table 7.15](#) lists the variable arguments that the compiler supports.

Table 7.15 Variable arguments (stdarg.h)

Function	Purpose
<code>va_arg(_ap, _type) (*(_type*)(_ap) -= sizeof(_type))</code>	Returns next parameter in argument list
<code>va_end(_ap) (void) 0</code>	Performs cleanup of argument list
<code>va_list</code>	Type declaration of variable argument list
<code>va_start(_ap, _parmN) (void) (_ap = (char*)&_parmN)</code>	Performs initialization of argument list

Standard Definitions (stddef.h)

[Table 7.16](#) lists the standard definitions that the compiler supports.

Table 7.16 Standard definitions (stddef.h)

Function	Purpose
<code>NULL ((void*) 0)</code>	Null pointer constant
<code>offsetof(type, member)</code>	Field offset in bytes from start of structure
<code>typedef int ptrdiff_t</code>	Signed integer type resulting from the subtraction of two pointers
<code>typedef int size_t</code>	Unsigned integer type that is the data type of the <code>sizeof</code> operator
<code>typedef short wchar_t</code>	Wide character type, as defined in ISO C

I/O Library (**stdio.h**)

The `stdio.h` library contains the following function types:

- Input Functions
- Stream functions
- Output functions
- Miscellaneous I/O functions

Input Functions

[Table 7.17](#) lists the input functions that the compiler supports.

Table 7.17 Input functions

Function	Purpose
<code>int fgetc(FILE*)</code>	Inputs a single character if available from specified stream
<code>size_t fread(void*, size_t, size_t, FILE*)</code>	Inputs a size number of characters from <code>stdin</code>
<code>int fscanf(FILE*, const char*, ...)</code>	Inputs text from the specified stream
<code>int getc(FILE*)</code>	Inputs a single character if available from specified stream
<code>int getchar(void)</code>	Inputs a single character if available from <code>stdin</code>
<code>int scanf(const char*, ...)</code>	Inputs text from <code>stdin</code>
<code>int sscanf(const char*, const char*, ...)</code>	Inputs text from specified string

Stream Functions

[Table 7.18](#) lists the stream functions that the compiler supports.

Table 7.18 Stream functions

Function	Purpose
<code>void clearerr(FILE*)</code>	Clears the EOF and error indicators for the specified stream
<code>int fclose(FILE*)</code>	Flushes the specified stream and closes the file associated with it
<code>int feof(FILE*)</code>	Tests the EOF indicator for the specified stream

Table 7.18 Stream functions (*continued*)

Function	Purpose
<code>int ferror(FILE*)</code>	Tests the error indicator for the specified stream
<code>int fgetpos(FILE*, fpos_t*)</code>	Stores the current value of the file position indicator for the specified stream
<code>FILE* freopen(const char*,const char*,FILE*)</code>	Opens the specified file in the specified mode, using the specified stream
<code>int fseek(FILE*, long int, int)</code>	Sets the file position indicator for the specified stream
<code>int fsetpos(FILE*, const fpos_t*)</code>	Sets the file position indicator for the specified stream to the specified value
<code>long int ftell(FILE*)</code>	Retrieves the current value of the file position indicator for the current stream
<code>int remove(const char*)</code>	Makes the specified file unavailable by its defined name
<code>int rename(const char*, const char*)</code>	Assigns to the specified file a new filename
<code>void rewind(FILE*)</code>	Sets the file position indicator for the specified stream to the beginning of the file
<code>void setbuf(FILE*, char*)</code>	Defines a buffer and associates it with the specified stream. A restricted version of <code>setvbuf()</code>
<code>int setvbuf(FILE*, char*, int, size_t)</code>	Defines a buffer and associates it with the specified stream
<code>stderr</code>	Standard error stream (Value = 3)
<code>stdin</code>	Standard input stream (Value = 1)
<code>stdout</code>	Standard output stream (Value = 2)
<code>FILE* tmpfile(void)</code>	Creates a temporary file
<code>char* tmpnam(char*)</code>	Generates a valid filename, meaning a filename that is not in use, as a string

Output Functions

[Table 7.19](#) lists the output functions that the compiler supports.

Table 7.19 Output functions

Function	Purpose
<code>char* fgets(char*, int, FILE*)</code>	Outputs characters to the specified stream
<code>int fprintf(FILE*, const char*, ...)</code>	Outputs the specified text to the specified stream
<code>int fputc(int, FILE*)</code>	Outputs a single character to the specified stream
<code>int fputs(const char*, FILE*)</code>	Outputs a string to the specified stream
<code>size_t fwrite(const void*, size_t, size_t, FILE*)</code>	Outputs a size number of characters to <code>stdout</code>
<code>char* gets(char*)</code>	Outputs characters into the user's buffer
<code>void perror(const char*)</code>	Outputs an error message
<code>int printf(const char*, ...)</code>	Outputs the specified text to <code>stdout</code>
<code>int putc(int, FILE*)</code>	Outputs a single character to the specified stream
<code>int putchar(int)</code>	Outputs a single character
<code>int puts (const char*)</code>	Outputs the string to <code>stdout</code> , followed by a newline
<code>int sprintf(char*, const char*, ...)</code>	Outputs the specified text to the specified buffer
<code>int vfprintf(FILE*, const char*, va_list)</code>	Outputs the variable arguments to the specified stream
<code>int vprintf(const char*, va_list)</code>	Outputs the variable arguments to <code>stdout</code>
<code>int vsprintf(char*, const char*, va_list)</code>	Outputs the variable arguments to the specified buffer

Miscellaneous I/O Functions

[Table 7.20](#) lists the miscellaneous I/O functions that the compiler supports.

Table 7.20 Miscellaneous I/O functions

Function	Purpose
<code>int fflush(FILE*)</code>	Causes the output buffers to be emptied to their destinations
<code>FILE* fopen(const char*, const char*)</code>	Associates a stream with a file
<code>int ungetc(int, FILE*)</code>	Moves the character back to the head of the input stream

General Utilities (stdlib.h)

The `stdlib.h` library contains the following function types:

- Memory allocation functions
- Integer arithmetic functions
- String conversion functions
- Searching and sorting functions
- Pseudo random number generation functions
- Environment functions
- Multibyte functions

Memory Allocation Functions

[Table 7.21](#) lists the memory allocation functions that the compiler supports.

Table 7.21 Memory allocation functions

Function	Purpose
<code>void free(void*)</code>	Returns allocated space to heap
<code>void* calloc(size_t, size_t)</code>	Allocates heap space initialized to zero
<code>void* malloc(size_t)</code>	Allocates heap space
<code>void* realloc(void*, size_t)</code>	Allocates a larger heap space and returns previous space to heap

Integer Arithmetic Functions

[Table 7.22](#) lists the integer arithmetic functions that the compiler supports.

Table 7.22 Integer arithmetic functions

Function	Purpose
<code>int abs(int)</code>	Absolute value
<code>div_t div(int, int)</code>	Quotient and remainder
<code>long int labs(long int)</code>	Computes absolute value and returns as long int
<code>ldiv_t ldiv(long int, long int)</code>	Quotient and remainder of long int

String Conversion Functions

[Table 7.23](#) lists the string conversion functions that the compiler supports.

Table 7.23 String conversion functions

Function	Purpose
<code>double atof(const char*)</code>	String to float
<code>int atoi(const char*)</code>	String to int
<code>long int atol(const char*)</code>	Long
<code>double strtod(const char*, char**)</code>	Double
<code>long int strtol(const char*, char**, int)</code>	Long
<code>unsigned long int strtoul(const char*, char**, int)</code>	Unsigned long

Searching and Sorting Functions

[Table 7.24](#) lists the searching and sorting functions that the compiler supports.

Table 7.24 Searching and sorting functions

Function	Purpose
<code>void *bsearch(const void*, const void*, size_t, size_t, int(*)(const void*, const void*))</code>	Binary search
<code>void *qsort(void*, size_t, size_t, int(*)(const void*, const void*))</code>	Quick sort

Pseudo Random Number Generation Functions

[Table 7.25](#) lists the pseudo random number generation functions that the compiler supports.

Table 7.25 Pseudo random number generation functions

Function	Purpose
<code>int rand(void)</code>	Random number generator
<code>void srand(unsigned int)</code>	Initializes the random number generator

Environment Functions

[Table 7.26](#) lists the environment functions that the compiler supports.

Table 7.26 Environment functions

Function	Purpose
<code>void abort(void)</code>	Causes an abnormal termination.
<code>int atexit(void (*)(void))</code>	Registers a function to be called at normal termination.
<code>void exit(int)</code>	Causes a normal termination
<code>char *getenv(const char *name)</code>	Gets environment variable. (This function is supported for compatibility purposes and has no effect.)
<code>int system(const char *string)</code>	Passes command to host environment. (This function is supported for compatibility purposes and has no effect.)

Multibyte Character Functions

[Table 7.27](#) lists the multibyte character functions that the compiler supports.

Table 7.27 Multibyte character functions

Function	Purpose
<code>int mblen(const char*, size_t)</code>	Multibyte string length
<code>size_t mbstowcs(wchar_t*, const char*, size_t)</code>	Converts multibyte string to wide character string
<code>int mbtowc(wchar_t*, const char*, size_t)</code>	Converts multibyte to wide character
<code>int wctomb(char*, wchar_t)</code>	Converts wide character to multibyte
<code>size_t wcstombs (char*, const wchar_t*, size_t)</code>	Converts wide character string to multibyte string

String Functions (string.h)

The `string.h` library contains the following function types:

- Copying functions
- Concatenation functions
- Comparison functions
- Search functions
- Other functions

Copying Functions

[Table 7.28](#) lists the copying functions that the compiler supports.

Table 7.28 Copying functions

Function	Purpose
<code>void* memcpy(void*, const void*, size_t)</code>	Copies data
<code>void* memmove(void*, const void*, size_t)</code>	Swaps data
<code>char* strcpy(char*, const char*)</code>	Copies a string
<code>char* strncpy(char*, const char*, size_t)</code>	Copies a string of a maximum length

Concatenation Functions

[Table 7.29](#) lists the concatenation functions that the compiler supports.

Table 7.29 Concatenation functions

Function	Purpose
<code>char* strcat(char*, const char*)</code>	Concatenates a string to the end of another string
<code>char* strncat(char*, const char*, size_t)</code>	Concatenates a string of specified maximum length to the end of another string

Comparison Functions

[Table 7.30](#) lists the comparison functions that the compiler supports.

Table 7.30 Comparison functions

Function	Purpose
<code>int memcmp(const void*, const void*, size_t)</code>	Compares data
<code>int strcmp(const char*, const char*)</code>	Compares strings
<code>int strcoll(const char*, const char*)</code>	Compares strings based on locale
<code>int strncmp(const char*, const char*, size_t)</code>	Compares strings of maximum length
<code>size_t strxfrm(char*, const char*, size_t)</code>	Transforms a string into a second string of the specified size

Search Functions

[Table 7.31](#) lists the search functions that the compiler supports.

Table 7.31 Search functions

Function	Purpose
<code>void* memchr(const void*, int, size_t)</code>	Searches for a value in the first number of characters
<code>char* strchr(const char*, int)</code>	Searches a string for the first occurrence of <code>char</code>
<code>size_t strcspn(const char*, const char*)</code>	Searches a string for the first occurrence of <code>char</code> in string set and returns the number of characters skipped
<code>char* strpbrk(const char*, const char*)</code>	Searches a string for the first occurrences of <code>char</code> in string set and returns a pointer to that location
<code>char* strrchr(const char*, int)</code>	Searches a string for the last occurrence of <code>char</code>

Runtime Libraries

Other Functions

Table 7.31 Search functions

Function	Purpose
<code>size_t strspn(const char*, const char*)</code>	Searches a string for the first occurrence of <code>char</code> not in string set
<code>char* strstr(const char*, const char*)</code>	Searches a string for the first occurrence of string
<code>char* strtok(char*, const char*)</code>	Separates a string into tokens

Other Functions

[Table 7.32](#) lists the other functions that the compiler supports.

Table 7.32 Other functions

Function	Purpose
<code>void* memset(void*, int, size_t)</code>	Copies a value into each number of characters
<code>char* strerror(int)</code>	Returns string for associated error condition
<code>size_t strlen(const char*)</code>	Returns size of string

Time Functions (time.h)

[Table 7.33](#) lists the time functions that the compiler supports.

Table 7.33 Time functions

Function	Purpose
<code>char *asctime(const struct tm *timeptr)</code>	Converts time to ASCII representation
<code>clock_t clock()</code>	Returns processor time
<code>typedef unsigned long clock_t</code>	Type used for measuring time
<code>char *ctime (const time_t *timer)</code>	Converts time to ASCII representation
<code>double difftime(time_t time1, time_t time0)</code>	Returns difference in seconds
<code>time_t mktime(struct tm *timeptr)</code>	Converts <code>struct tm</code> to <code>time_t</code>
<code>size_t strftime (char *s, size_t maxsize, const char *format, const struct tm *timeptr)</code>	Converts an ASCII string to <code>time_t</code>
<code>time_t time(time_t *timer)</code>	Returns processor time (same as <code>clock</code>)

Table 7.33 Time functions

Function	Purpose
<code>typedef unsigned long time_t</code>	Type used for measuring time
<code>struct tm *gmtime(const time_t *timer)</code>	Returns time in GMT time zone
<code>struct tm *localtime(const time_t *timer)</code>	Returns time in local time zone

Time Constant

[Table 7.34](#) shows the time constant that the compiler supports.

Table 7.34 Time constant

Constant	Value	Purpose
<code>CLOCKS_PER_SEC</code>	TBD	

Process Time

The `clock` function returns the current value of the system timer. This function must be configured to match the actual system timer configuration. The timer is started and set for a maximum period during the initialization of any C program that references the `clock` function, and is used only by this function. The return value of `clock` has type `clock_t`, which is unsigned long.

[Listing 7.5](#) shows how to use the `clock` function to time your application.

Listing 7.5 Timing an application

```
#include <time.h>
clock_t start, end, elapsed;

/* . . . application setup . . . */

start = clock( );

/* . . . application processing . . . */

end = clock( );

elapsed = end - start; /* Assumes no wrap-around */
```

```
printf("Elapsed time: %Lu * 2 cycles. \n", elapsed);
```

Built-in Intrinsic Functions (prototype.h)

The compiler supports a set of built-in intrinsic functions that enable fractional operations to be implemented using integer data types, by mapping directly to SC100 assembly instructions.

[Table 7.35](#) lists these built-in intrinsic functions.

Table 7.35 Built-in intrinsic functions

Function	Purpose
short abs_s(short var1)	Short absolute value of var1. For example, the result of abs_s(-32768) is +32767.
short add(short var1,short var2)	Short add. Performs the addition var1+var2 with overflow control and saturation. The 16-bit result is set at +32767 when overflow occurs, or at -32768 when underflow occurs.
BitReverseUpdate	Increments the iterator with bit reverse.
Word64 D_add(Word64 D_var1,Word64 D_var2)	Double precision add. Performs the addition D_var1+D_var2 with overflow control and saturation.
long D_extract_h(Word64 D_var1)	Double precision extract high. Returns the 32 MSB of the 64-bit value D_var1.
unsigned long D_extract_l (Word64 D_var1)	Double precision extract low. Returns the 32 LSB of the 64-bit value D_var1 as an unsigned 32-bit value.
Word64 D_mac(Word64 D_var3, long L_var1, long L_var2)	Double precision multiply accumulate. Multiplies L_var1 by L_var2 and shifts the result left by 1. Adds the 64-bit result to L_var3 with saturation, and returns a 64-bit result. For example: D_mac(D_var3,L_var1,L_var2) = D_add(D_var3,D_mult(L_var1,L_var2))
Word64 D_msu(Word64 D_var3, long L_var1, long L_var2)	Double precision multiply subtract. Multiplies L_var1 by L_var2 and shifts the result left by 1. Subtracts the 64-bit result from D_var3 with saturation, and returns a 64-bit result. For example: D_msu(D_var3,L_var1,L_var2) = D_sub(D_var3,D_mult(L_var1,L_var2))

Table 7.35 Built-in intrinsic functions (*continued*)

Function	Purpose
Word64 D_mult(long L_var1, long L_var2)	Double precision multiply. The 64-bit result of the multiplication of L_var1 by L_var2 with one shift left. For example: D_mult(L_var1,L_var2) = D_shl((L_var1*L_var2),1)
long D_round(Word64 D_var1)	Double precision round. Rounds the lower 32 bits of the 64-bit D_var1 into the MS 32 bits with saturation. Shifts the resulting bits right by 32 and returns the 32-bit value.
Word64 D_sat(Word64 D_var1)	Double precision saturation. Saturates a 64-bit value.
Word64 D_set(long L_var1, unsigned long L_var2)	Double precision concatenation. Concatenates two 32-bit values, L_var1 and unsigned L_var2, into one 64-bit value.
Word64 D_sub(Word64 D_var1, Word64 D_var2)	Double precision subtract. 64-bit subtraction of the two 64-bit variables (D_var1-D_var2) with overflow control and saturation.
void debug()	Generates assembly instruction to enter Debug mode.
void debugev()	Generates assembly instruction to issue Debug event.
void di()	Generates assembly instruction to disable interrupts.
short div_s(short var1,short var2)	Short divide. Produces a result which is the fractional integer division of var1 by var2; var1 and var2 must be positive, and var2 must be greater or equal to var1. The result is positive (leading bit equal to 0) and truncated to 16 bits. If var1 = var2 then div(var1,var2) = 32767.
void ei()	Generates assembly instruction to enable interrupts.
EndBitReverse	Frees bit reverse iterator.
short extract_h(long L_var1)	Long extract high. Returns the 16 MSB of L_var1.
short extract_l(long L_var1)	Long extract low. Returns the 16 LSB of L_var1.
void illegal()	Generates assembly instruction to execute illegal exception.
InitBitReverse	Allocates a bit reverse iterator.
long L_abs(long L_var1)	Long absolute value of L_var1. Saturates in cases where the value is -214783648.

Runtime Libraries

Built-in Intrinsic Functions (*prototype.h*)

Table 7.35 Built-in intrinsic functions (*continued*)

Function	Purpose
<code>long L_add(long L_var1, long L_var2)</code>	<p>Long add. 32-bit addition of the two 32-bit variables (<code>L_var1+L_var2</code>) with overflow control and saturation.</p> <p>The result is set at +2147483647 when overflow occurs, or at -2147483648 when underflow occurs.</p>
<code>long L_deposit_h(short var1)</code>	Deposit short in MSB. Deposits the 16-bit <code>var1</code> into the 16 MS bits of the 32-bit output. The 16 LS bits of the output are zeroed.
<code>long L_deposit_l(short var1)</code>	Deposit short in LSB. Deposits the 16-bit <code>var1</code> into the 16 LS bits of the 32-bit output. The 16 MS bits of the output are sign extended.
<code>long L_mac(long L_var3, short var1, short var2)</code>	<p>Multiply accumulate. Multiplies <code>var1</code> by <code>var2</code> and shifts the result left by 1. Adds the 32-bit result to <code>L_var3</code> with saturation, and returns a 32-bit result. For example:</p> <p><code>L_mac(L_var3, var1, var2) = L_add(L_var3, L_mult(var1, var2))</code></p>
<code>long L_max(long L_var1, long L_var2)</code>	Compares the values of two 32-bit variables and returns the higher value of the two.
<code>long L_min(long L_var1, long L_var2)</code>	Compares the values of two 32-bit variables and returns the lower value of the two.
<code>long L_msu(long L_var3, short var1, short var2)</code>	<p>Multiply subtract. Multiplies <code>var1</code> by <code>var2</code> and shifts the result left by 1. Subtracts the 32-bit result from <code>L_var3</code> with saturation, and returns a 32-bit result. For example:</p> <p><code>L_msu(L_var3, var1, var2) = L_sub(L_var3, L_mult(var1, var2))</code></p>
<code>long L_mult(short var1, short var2)</code>	<p>Long multiply. The 32-bit result of the multiplication of <code>var1</code> by <code>var2</code> with one shift left, for example:</p> <p><code>L_mult(var1, var2) = L_shl((var1*var2), 1)</code></p> <p>and</p> <p><code>L_mult(-32768, -32768) = 2147483647</code></p>
<code>long L_negate(long L_var1)</code>	Long negate. Negates the 32-bit variable <code>L_var1</code> with saturation. Saturates in cases where the value is -2147483648 (0x8000 0000).

Table 7.35 Built-in intrinsic functions (*continued*)

Function	Purpose
<code>long L_rol(long L_var1)</code>	Long rotate left. Rotates the 32-bit variable <code>L_var1</code> left into a 40-bit value, and returns a 32-bit result..
<code>long L_ror(long L_var1)</code>	Long rotate right. Rotates the 32-bit variable <code>L_var1</code> right into a 40-bit value, and returns a 32-bit result.
<code>long L_sat(long L_var1)</code>	Saturates a 32-bit value.
<code>long L_shl(long L_var1, short var2)</code>	Long shift left. Arithmetically shifts the 32-bit <code>L_var1</code> left <code>var2</code> positions. Zero fills the <code>var2</code> LSB of the result. If <code>var2</code> is negative, arithmetically shifts <code>L_var1</code> right by <code>var2</code> with sign extension. Saturates the result in cases where underflow or overflow occurs.
<code>long L_shr(long L_var1, short var2)</code>	Long shift right. Arithmetically shifts the 32-bit <code>L_var1</code> right <code>var2</code> positions with sign extension. If <code>var2</code> is negative, arithmetically shifts <code>L_var1</code> left by <code>var2</code> and zero fills the <code>var2</code> LSB of the result. Saturates the result in cases where underflow or overflow occurs.
<code>long L_shr_r(long L_var1, short var2)</code>	Long shift right and round. Same as <code>L_shr(L_var1, var2)</code> but with rounding. Saturates the result in cases where underflow or overflow occurs.
<code>long L_sub(long L_var1, long L_var2)</code>	Long subtract. 32-bit subtraction of the two 32-bit variables (<code>L_var1 - L_var2</code>) with overflow control and saturation. The result is set at +2147483647 when overflow occurs or at -2147483648 when underflow occurs.
<code>short mac_r(long L_var3, short var1, short var2)</code>	Multiply accumulate and round. Multiplies <code>var1</code> by <code>var2</code> and shifts the result left by 1. Adds the 32-bit result to <code>L_var3</code> with saturation. Rounds the LS 16 bits of the result into the MS 16 bits with saturation and shifts the result right by 16. Returns a 16-bit result.
<code>void mark()</code>	Generates assembly instruction to write program counter to trace buffer, if trace buffer enabled.
<code>short max(short var1, short var2)</code>	Compares the values of two 16-bit variables and returns the higher value of the two.
<code>short min(short var1, short var2)</code>	Compares the values of two 16-bit variables and returns the lower value of the two.

Runtime Libraries

Built-in Intrinsic Functions (*prototype.h*)

Table 7.35 Built-in intrinsic functions (*continued*)

Function	Purpose
<code>long mpyuu(long L_var1, long L_var2)</code>	Multiplies the 16 LSB of two 32-bit variables, treating both variables as unsigned values, and returns a 32-bit result.
<code>long mpyus(long L_var1, long L_var2)</code>	Multiplies the 16 LSB of the 32-bit variable <code>L_var1</code> , treated as an unsigned value, by the 16 MSB of the 32-bit variable <code>L_var2</code> , treated as a signed value. Returns a 32-bit result.
<code>long mpysu(long L_var1, long L_var2)</code>	Multiplies the 16 MSB of the 32-bit variable <code>L_var1</code> , treated as a signed value, by the 16 LSB of the 32-bit variable <code>L_var2</code> , treated as an unsigned value. Returns a 32-bit result.
<code>short msu_r(long L_var3, short var1, short var2)</code>	Multiply subtract and round. Multiplies <code>var1</code> by <code>var2</code> and shifts the result left by 1. Subtracts the 32-bit result from <code>L_var3</code> with saturation. Rounds the LS 16 bits of the result into the MS 16 bits with saturation and shifts the result right by 16. Returns a 16-bit result.
<code>short mult(short var1,short var2)</code>	Short multiply. Performs the multiplication of <code>var1</code> by <code>var2</code> and gives a 16-bit result which is scaled. For example: <code>mult(var1,var2) = extract_l(L_shr((var1 * var2),15))</code> and <code>mult(-32768,-32768) = 32767</code>
<code>short mult_r(short var1, short var2)</code>	Multiply and round. Same as <code>mult</code> with rounding. For example: <code>mult_r(var1,var2) = extract_l (L_shr(((var1*var2)+16384),15))</code> and <code>mult_r(-32768,-32768) = 32767.</code>
<code>short negate(short var1)</code>	Short negate. Negates <code>var1</code> with saturation. Saturates in cases where the value is -32768. For example: <code>negate(var1) = sub(0,var1).</code>

Table 7.35 Built-in intrinsic functions (*continued*)

Function	Purpose
<code>short norm_l(long L_var1)</code>	<p>Normalizes any long fractional value. Produces the number of left shifts needed to normalize the 32-bit variable <code>L_var1</code> for positive values on the interval with minimum of 1073741824 and maximum of 2147483647, and for negative values on the interval with minimum of -2147483648 and maximum of -1073741824. In order to normalize the result, the following operation must be executed:</p> <pre>norm_L_var1 = L_shl(L_var1,norm_l(L_var1))</pre>
<code>short norm_s(short var1)</code>	<p>Normalizes any fractional value. Produces the number of left shifts needed to normalize the 16-bit variable <code>var1</code> for positive values on the interval with minimum of 16384 and maximum of 32767, and for negative values on the interval with minimum of -32768 and maximum of -16384. In order to normalize the result, the following operation must be executed:</p> <pre>norm_var1 = shl(var1,norm_s(var1)).</pre>
<code>short round(long var1)</code>	<p>Round. Rounds the lower 16 bits of the 32-bit number into the MS 16 bits with saturation. Shifts the resulting bits right by 16 and returns the 16-bit number. For example:</p> <pre>round(L_var1) = extract_h(L_add(L_var1,32768))</pre>
<code>short saturate(short var1)</code>	Saturates a 16-bit value.
<code>setcnvrm()</code>	Sets rounding mode to convergent rounding mode.
<code>set2crm()</code>	Sets rounding mode to two's-complement rounding mode.
<code>void setnosat()</code>	Clears saturation mode bit in status register.
<code>void setsat32()</code>	Sets saturation mode bit in status register.
<code>short shl(short var1,short var2)</code>	<p>Short shift left. Arithmetically shifts the 16-bit <code>var1</code> left <code>var2</code> positions. Zero fills the <code>var2</code> LSB of the result. If <code>var2</code> is negative, arithmetically shifts <code>var1</code> right by <code>var2</code> with sign extension. Saturates the result in cases where underflow or overflow occurs.</p>

Runtime Libraries

Built-in Intrinsic Functions (*prototype.h*)

Table 7.35 Built-in intrinsic functions (*continued*)

Function	Purpose
<code>short shr(short var1, short var2)</code>	Short shift right. Arithmetically shifts the 16-bit <code>var1</code> right <code>var2</code> positions with sign extension. If <code>var2</code> is negative, arithmetically shifts <code>var1</code> left by <code>var2</code> with sign extension. Saturates the result in cases where underflow or overflow occurs.
<code>short shr_r(short var1, short var2)</code>	Short shift right and round. Same as <code>shr(var1, var2)</code> but with rounding. Saturates the result in cases where underflow or overflow occurs.
<code>void stop()</code>	Generates assembly instruction to enter <code>Stop</code> low power mode.
<code>short sub(short var1, short var2)</code>	Performs the subtraction with overflow control and saturation. The 16-bit result is set at +32767 when overflow occurs or at -32768 when underflow occurs.
<code>void trap()</code>	Generates assembly instruction to execute <code>Trap</code> exception.
<code>void wait()</code>	Generates assembly instruction to enter <code>Wait</code> low power mode.
<code>Word40 X_abs(Word40 X_var1)</code>	40-bit absolute value of <code>X_var1</code> .
<code>Word40 X_add(Word40 X_var1, Word40 X_var2)</code>	Extended precision add. Performs the addition <code>X_var1+X_var2</code> without saturation.
<code>Word40 X_extend(long L_var1)</code>	Sign extend 32-bit value to 40-bit value.
<code>short X_extract_h(Word40 X_var1)</code>	Extended precision extract high. Returns the 16 MSB of the 40-bit value <code>X_var1</code> .
<code>short X_extract_l(Word40 X_var1)</code>	Extended precision extract low. Returns the 16 LSB of the 40-bit value <code>X_var1</code> .
<code>Word40 X_mac(Word40 X_var3, short var1, short var2)</code>	Extended precision multiply accumulate. Multiplies <code>var1</code> by <code>var2</code> and shifts the result left by 1. Adds the 40-bit result to <code>X_var3</code> without saturation, and returns a 40-bit result. For example: <code>X_mac(X_var3, var1, var2) = X_add(X_var3, X_mult(var1, var2))</code>
<code>Word40 X_msu(Word40 X_var3, short var1, short var2)</code>	Extended precision multiply subtract. Multiplies <code>var1</code> by <code>var2</code> and shifts the result left by 1. Subtracts the 40-bit result from <code>var3</code> without saturation, and returns a 40-bit result. For example: <code>X_msu(X_var3, var1, var2) = X_sub(X_var3, X_mult(var1, var2))</code>

Table 7.35 Built-in intrinsic functions (*continued*)

Function	Purpose
Word40 X_mult(short var_1, short var_2)	Extended precision multiply. The 40-bit result of the multiplication of var1 by var2 with one shift left, for example: X_mult(var1, var2) = X_shl((var1*var2), 1)
short X_norm(Word40 X_var1)	Normalizes a 40-bit fractional value.
Word40 X_or(Word40 X_var1, Word40 X_var2)	Performs logical OR on two 40-bit values.
Word40 X_rol(Word40)	Rotates left a 40-bit value.
Word40 X_ror(Word40)	Rotates right a 40-bit value.
short X_round(Word40 X_var1)	Extended precision round. Rounds the lower 16 bits of the 40-bit number into the MS 16 bits without saturation. Shifts the resulting bits right by 16 and returns the 16-bit number.
long X_sat(Word40 X_var1)	Extended precision saturation. Saturates a 40-bit value.
Word40 X_set(char var1, unsigned long L_var2)	Extended precision concatenation. Concatenates an 8-bit character value and an unsigned 32-bit value into one 40-bit value.
Word40 X_shl(Word40 X_var1, short var2)	Extended shift left. Arithmetically shifts the 40-bit X_var1 left var2 positions. Zero fills the var2 LSB of the result. If var2 is negative, arithmetically shifts X_var1 right by var2 with sign extension.
Word40 X_shr(Word40 X_var1, short var2)	Extended shift right. Arithmetically shifts the 40-bit X_var1 right var2 positions with sign extension. If var2 is negative, arithmetically shifts X_var1 left by var2 and zero fills the var2 LSB of the result.
Word40 X_sub(Word40 X_var1, Word40 X_var1)	Extended precision subtract. 40-bit subtraction of the two 40-bit variables (X_var1-X_var2) without saturation.
long X_trunc(Word40 X_var1)	Truncates 40-bit value into 32-bit value.
void trap();	Calls the trap instruction.
void trap_r(void *);	Stores its argument in \$r0 and calls the trap instruction. Side effects are assumed on the argument pointer.
int trap_d(int);	Stores its argument in \$d0 and calls the trap instruction.

Runtime Libraries

Built-in Intrinsic Functions (*prototype.h*)

Table 7.35 Built-in intrinsic functions (*continued*)

Function	Purpose
<code>int readSR(void);</code>	Returns the content of the status register.
<code>void writeSR(int);</code>	Writes the status register with the passed value.
<code>void setPPL(int);</code>	This function assumes its argument is always an immediate value. This function outperforms: <ul style="list-style-type: none">• disable interrupts• clear bits 23 to 21 in \$sr register (Interrupt mask bits)• set bits 23 to 21 in \$sr register to argument value• enable interrupts
<code>Word32 mpyus_shr16(Word32 var1, Word16 var2);</code>	Performs the unsigned signed fractional multiplication of the lsb of <code>var1</code> by <code>var2</code> and shifts the result by 16 to the right.
<code>Word32 mpysu_shr16(Word16 var1, Word32 var2);</code>	Performs the signed unsigned fractional multiplication of the lsb of <code>var2</code> by <code>var1</code> and shifts the result by 16 to the right.
<code>Word32 L_mult_ls(Word32 var1, Word16 var2);</code>	Performs a 32*16 fractional multiplication. Be aware that this is an optimized version whose results differ from the regular 32*16 mult in the least significant bit (2**-31 error). An example follows: <code>L_mult_ls(X, Y) = dmac_ss(mpy_us(X, Y), X, Y);</code>
<code>Word32 L_mult_sl(Word16 var2, Word32 var1);</code>	Performs a 16*32 fractional multiplication. Be aware that this is an optimized version whose results differ from the regular 16*32 mult in the least significant bit (2**-31 error). An example follows: <code>L_mult_sl(X, Y) = dmac_ss(mpy_us(Y, X), Y, X);</code>
<code>void Set_Overflow(Word32 Value);</code>	Sets overflow flag in sr to 1 if <code>Value</code> does not equal 0 or to 0 if value is equal to 0. Examples follow: <code>Set_Overflow(1);</code> <code>Set_Overflow(0);</code> <code>Set_Overflow(X);</code>
<code>int Get_Overflow();</code>	Returns the current value of the overflow bit in sr. An example follows: <code>X = Get_Overflow();</code>

Table 7.35 Built-in intrinsic functions (*continued*)

Function	Purpose
<code>int Test_Overflow();</code>	Always use this function in a conditional expression (for example): <pre>if (Test_Overflow()) { printf("There is an overflow\n");</pre> <p>(A statement like <code>X = Test_Overflow()</code> results in an error.)</p>
<code>Word40 X_xor (Word40 var1, Word40 var2);</code>	Returns the xor on 40 bits of <code>var1</code> and <code>var2</code> .
<code>Word32 X_sat (Word40 var1);</code>	Returns <code>var1</code> as a saturated Q31 number.
<code>Word16 shl_nosat (Word16 var1, Word16 var2);</code>	Assumes <code>var1 << var2</code> where <code>var1</code> is a Q15 do not saturate, therefore avoiding the saturation check (useful for normalization).
<code>Word32 L_shl_nosat (Word32 L_var1, Word16 var2);</code>	Assumes <code>L_var1 << var2</code> where <code>var1</code> is a Q31 do not saturate, therefore avoiding the saturation check (useful for normalization).
<code>Word16 shr_nosat (Word16 var1, Word16 var2);</code>	Assumes <code>var1 >> var2</code> where <code>var1</code> is a Q15 do not saturate therefore avoiding the saturation check (useful for normalization).
<code>Word32 L_shr_nosat (Word32 L_var1, Word16 var2);</code>	Assumes <code>L_var1 >> var2</code> where <code>var1</code> is a Q31 do not saturate therefore avoiding the saturation check (useful for normalization).
<code>Word16 neg_norm_s (Word16 var1);</code>	Returns the count of the leading zero bit of <code>var1</code> (a Q15 number) as a negative number. Maps straight to the sc1400 <code>clb</code> instruction.
<code>Word16 neg_norm_l (Word32 L_var1);</code>	Returns the count of the leading zero bit of <code>L_var1</code> (a Q31 number) as a negative number. Maps straight to the sc1400 <code>clb</code> instruction.

Runtime Libraries

Built-in Intrinsic Functions (prototype.h)

Migrating from Other Environments

The Metrowerks Enterprise C compiler provides header files that make it easy to migrate C code developed for certain other compilers. The compilation and its results may be affected in various ways by the differences between specific compiler environments and the compiler. The effects may include, for example, assembler errors for inlined code that is not supported, or loss of efficiency for functions that are supported, but implemented in a different way.

This appendix contains the following topics:

- [Code Migration Overview](#)
- [Migrating Code Developed for DSP56600](#)
- [Migrating Code Developed for TI6xx](#)

Code Migration Overview

In most circumstances, the compiler can successfully compile standard ANSI code that:

- Does not use compiler-specific extensions
- Does not rely implicitly on the sizes of data types
- Does not rely on system-specific features, such as memory maps or peripherals
- Does not rely on undefined compiler behavior

The compiler runtime libraries include a header file for each environment for which code is accepted, as follows:

- DSP56600 compilers: `port566toSC1.h` header file
- TI6xx compilers: `portc6xtoSC1.h` header file

The features used in the specified environment are defined in the relevant header file with correct values, to ensure that the code is not affected and compiles successfully.

To use these definitions, just include the appropriate header file to your source code. For example, when migrating code from the DSP56600 compiler environment, include the `port566toSC1.h` header file, as shown in [Listing A.1](#).

Listing A.1 Migrating code from other environments

```
#include <port566toSC1.h>
void main()
{
}
```

Migrating Code Developed for DSP56600

This section discusses differences to consider when using the Metrowerks Enterprise C compiler with code developed for the DSP56600 family of compilers.

Integer Data Types

The DSP56600 and SC100 compilers map certain integer data types to different sizes. [Table A.1](#) lists the data type size discrepancies that relate to integers:

Table A.1 DSP56600 Integer Data Type Differences

Data Type	DSP56600 Compiler	SC100 C Compiler
char unsigned char	Saved in memory as 16 bits. Some operations are performed with 16 bits, others with 8.	8 bits
packed char	8 bits	Not supported
int unsigned int	16 bits	32 bits
enum	16 bits	32 bits

Fractional Data Types

DSP56600 compilers use built-in data types for declaring fractional variables. The Metrowerks Enterprise C compiler uses standard integer types for both fractional and integer values. [Table A.2](#) lists the fractional data type differences:

Table A.2 DSP56600 Fractional Data Type Differences

Data Type	DSP56600 Compiler	SC100 C Compiler
16-bit fraction	<code>_fract</code>	Word16
32-bit fraction	<code>long_fract</code>	Word32
40-bit accumulator	<code>long_fract</code>	Word40
64-bit fraction	Not supported	Word64
Complex fractions	<code>_complex</code>	Not supported directly

Floating Point Data Types

DSP56600 compilers represent floating point data types according to a 32-bit proprietary format. The Metrowerks Enterprise C compiler maps fractional data types to a single-precision IEEE-754 type, using 32 bits. As a result, there may be differences in the numerical accuracy of floating point calculations.

Pointers

The difference in pointer size between the two compilers is shown in [Table A.3](#):

Table A.3 DSP56600 Pointer Size Differences

Data Type	DSP56600 Compiler	SC100 C Compiler
pointer to char	16 bits	32 bits
pointer to short	16 bits	32 bits, even addresses only
pointer to long	16 bits	32 bits, quad addresses only

In most circumstances, the difference in pointer size is unlikely to have any impact, since the relevant addresses are usually mapped to different numerical values on different processors.

Fractional Arithmetic

DSP56600 compilers support fractional arithmetic using integer-like operators, such as the plus sign (+) and multiplication symbol (*). The Metrowerks Enterprise C compiler implements fractional operations through the use of intrinsic functions. [Table A.4](#) lists the DSP56600 fractional operations and shows the equivalent Metrowerks Enterprise C compiler intrinsic functions:

Table A.4 DSP56600 Fractional Arithmetic Differences

Fractional Operation	DSP56600 Compiler	SC100C Compiler
Addition	+	Word16 add Word32 L_add
Subtraction	-	Word16 sub Word32 L_sub
Absolute value	_fabs _lfabs	Word16 abs_s Word32 L_abs
Multiplication	*	Word16 mult Word32 L_mult Word16 mult_r
Shift right	>>	Word16 shr Word32 L_shr
Shift left	<<	Word16 shl Word32 L_shl
Negate	-	Word16 negate Word32 L_negate
Round	_fract_round	Word16 round
Divide	_pdiv	Word16 div_s
Normalize	Can be implemented using _asm	Word16 norm_s Word16 norm_l
Saturation control	Can be implemented using _asm	void setnosat void setsat32

The Metrowerks Enterprise C compiler supports many more fractional operations, including 40-bit and 64-bit fractional functions, which are not supported in the DSP56600 environment.

Inlined Assembly and C Code

DSP56600 compilers use `_inline` and `_asm` to designate a C routine for inlining, and to define the instructions, operands and modifiers for inlined assembly statements. The Metrowerks Enterprise C compiler uses the pragma `#pragma inline` to specify an inlined function.

Intrinsic Functions

The Metrowerks Enterprise C compiler library routines support a number of DSP56600 intrinsic functions, as shown in [Table A.5](#):

Table A.5 DSP56600 Intrinsic Function Differences

Description	DSP56600 Compiler	SC100 C Compiler
Bit field operations	<code>_bfchg()</code> <code>_bfclr()</code> <code>_bfset()</code> <code>_bftsth()</code> <code>_bftstl()</code>	Can be implemented by library routines
Cache control	<code>_cache_get_start()</code> <code>_cache_get_end()</code> <code>_pflush()</code> <code>_pflushun()</code> <code>_pfree()</code> <code>_plock()</code> <code>_punlock()</code>	Not available
Fraction to integer coercion	<code>_fract2int()</code> <code>_lfraact2long</code>	Not needed (both represented by integers)
Integer to fraction coercion	<code>_intt2fract()</code> <code>_long2lfract()</code>	Not needed (both represented by integers)
Extend byte in accumulator	<code>_ext()</code>	Not applicable
Fractional square root	<code>_fsqrt()</code>	Can be implemented by a library routine
String copy (inlined)	<code>_strcmp()</code>	Supported as a library routine (<code>strcmp</code>)
Absolute of long integer	<code>_labs</code>	<code>labs()</code>
Insert NOP instruction	<code>_nop()</code>	<code>_asm("nop")</code>
STOP instruction	<code>_stop()</code>	<code>stop()</code>
Software interrupt	<code>_swi()</code>	<code>trap()</code>
WAIT instruction	<code>_wait()</code>	<code>wait()</code>
Viterbi operation	<code>_vsl</code>	Can be implemented by a library routine

Pragmas

The functions of the DSP56600 inlined assembly pragmas `asm`, `asm_noflush` and `endasm` are supported by the Metrowerks Enterprise C compiler using a function qualifier. The Metrowerks Enterprise C compiler loop optimization pragma `#pragma loop_count` is the equivalent of the DSP56600 pragmas `iterate_at_least_once` and `no_iterate_at_least_once`.

The following DSP56600 pragmas have no equivalent in the Metrowerks Enterprise C compiler environment:

- `cache_align_now`
- `cache_sector_size`
- `cache_region_start`
- `cache_region_endpack_strings`
- `nopack_strings`
- `source`
- `nosource`
- `jumptable_memory`

Interrupt Handlers

The Metrowerks Enterprise C compiler pragma `interrupt` performs the function of both `_fast_interrupt` and `_long_interrupt` in the DSP56600 environment.

Storage Specifiers

The DSP56600 compilers support a number of storage specifiers, which are either not used in the SC100 environment, or are specified at link time, as shown in [Table A.6](#):

Table A.6 DSP56600 Storage Specifiers

Storage	DSP56600 Compiler	SC100 C Compiler
X memory	<code>_X</code>	Not applicable
Y memory	<code>_Y</code>	Not applicable
Program memory	<code>_P</code>	Not applicable
L memory	<code>_L</code>	Not applicable

Table A.6 DSP56600 Storage Specifiers

Storage	DSP56600 Compiler	SC100 C Compiler
Lowest 64 words in data memory	<code>_near</code>	Not applicable
Internal memory	<code>_internal</code>	Specified at link time
External memory	<code>_external</code>	Specified at link time
Absolute address for global variable	<code>_at</code>	Specified at link time in the application configuration file

Miscellaneous

[Table A.7](#) outlines some further differences between the two compilers:

Table A.7 DSP56600 Miscellaneous Differences

Description	DSP56600 Compiler	SC100 C Compiler
Wrap-around semantics for fractional data	<code>_nosat</code>	Not applicable
Force DSP56300 GNU calling convention	<code>_compatible</code>	Not applicable
Circular buffer support	<code>_circ</code>	Addressing calculations using the C modulo (%) operator

Migrating Code Developed for TI6xx

This section discusses differences to consider when using the compiler with code developed for the TI6xx family of compilers.

Data Types

TI6xx compilers map the integer type `long` to 40 bits. The compiler defines the integer type `long` as 32 bits. C code that relies on the fact that type `long` is 40 bits wide must be modified before it can be migrated.

Keywords

The TI6xx keywords `cregister`, `near` and `far` are not supported by the compiler. When including the migration header file, these

keywords are accepted but have no effect on the compilation results.

The TI6xx keywords `interrupt` and `inline` are supported, but are implemented differently, using `#pragma inline` and `#pragma interrupt`. As a result, no automatic translation is provided. The code must be modified to use the pragmas supported by the compiler.

Pragmas

TI6xx pragmas are ignored. Warnings are issued, but the correctness of the compilation is not affected.

Inlined Assembly Code

By definition, inlined assembly code is not portable from one environment to another. The SC100 Assembler is unable to recognize inlined TI6xx assembly code, and issues errors.

Intrinsic Functions

The TI6xx intrinsic functions listed in the `portc6xtoSC1.h` header file are supported. These are functionally equivalent to their corresponding TI6xx intrinsic functions, but their performance may be significantly affected.

Modulo Addressing Example

The modulo addressing support provides a fully functional C implementation regardless of the target or compiler. You can compile the example in this appendix using a simulator on a PC or a workstation and receive correct results.

When you compile the example on the SC140 with the Metrowerks Enterprise C compiler using the `-mod` option, the compiler tries to use the modulo addressing mode if it can prove that it is a valid usage.

For the usage to be valid, the modulo must be equivalent to a conditional subtraction reflecting the fact that modulo addressing is implemented as follows on the target:

```
if (Pointer > (Base + Mod)),  
    then Pointer = Pointer - Mod
```

Modulo access is optimally supported when offsets are used; the base can either be an array or a pointer.

[Listing B.1](#) shows ways of using modulo addressing when bases are pointers.

Listing B.1 Modulo Addressing Examples

```
int fct(short *pt, short *pt1, unsigned int Max)  
{  
    // First style, straightforward usage of the loop index  
    // combined with modulo  
  
    int i, j;  
    int Acc = 0;
```

Modulo Addressing Example

```
        for (i = j = 0; i<Max; i++, j++) {
            Acc = pt[i%3] + pt1[i%5];
        }

        return Acc;
    }

int fct_1(short *pt, short *pt1, unsigned int Max)

{
    // Second style: explicit update of the
    // index : index = (index + step) % buf_size
    //
    // This one will always work
    // if step < buf_size and if intial_value(index) < (pt + buf_size)

    int i, j, k;
    int Acc = 0;

    for (i = j = k = 0; i<Max; i++) {
        Acc = pt[j] + pt1[k];
        j = (j+1)%3; k = (k+2)%7;
    }

    return Acc;
}
```

```
;*****
; File Name :          ce2.sl
; Invocation line: /home/comtools/enterprise/new_prod/real-bin/
scc -mod -s -c ce2.c
;
;*****
        .file    "ce2.c"

        section .data local
            align    8
F__MemAllocArea
            align    4

        endsec

        section .text local
TextStart_ce2

bb_cs_offset__fct      equ    0          ; at __fct sp = 0
bb_cs_offset_DW_2      equ    2          ; at DW_2 sp = 2
bb_cs_offset_DW_20     equ    22         ; at DW_20 sp = 22
bb_cs_offset_DW_21     equ    20         ; at DW_21 sp = 20

;*****
;
; Function __fct
;
; Stack frame size: 48
;
; Calling Convention: Standard
;
; Parameter pt    passed in register r0
; Parameter pt1   passed in register r1
; Parameter Max   passed in stack with offset -12
;
; Returned value  ret_fct    passed in register d0
;
;*****
```

Modulo Addressing Example

```

        global  _fct
        align   16
_fct:    type   func
[
    clr        d0                ; [23]
    push       r6                ; [17]
    push       r7                ; [17]
]
DW_2:
[
    move.l     (sp-20), r2        ; [25]
    adda       #40, sp, r6       ; [0]
]
[
    tfra       r6, sp            ; [0]
    tfra       r0, r3           ; [26] B1
]
DW_4:
[
    tstega     r2                ; [25]
    move.w     #<6, m0           ; [0] B1
]
[
    bt         <L2               ; [25]
    move.l     (sp-60), d4       ; [0]
]
[
    move.w     #<10, m1          ; [0]
    bmset      #32768, mctl.l    ; [0]
]
[
    move.w     #<0, d0           ; [0]
    tfra       r0, r11          ; [0]
]
[
    max        d0, d4            ; [0]
    tfra       r1, r10          ; [0]
    bmset      #2304, mctl.l     ; [0]
]
[
    doensh3    d4                ; [0]
    tfra       r1, r2           ; [26]
]
[
    nop                ; [0] L_D_3
]
```

```

        nop                                ; [0] L_D_3
        loopstart3
L10
    [
        move.w    (r2)+,d1                ; [26]
        move.w    (r3)+,d0                ; [26]
    ]
        add       d0,d1,d0                ; [25]
        loopend3
        bmclr     #3840,mctl.l            ; [0]
        bmclr     #61440,mctl.l          ; [0]
L2
        move.w    #40,r6                  ; [29]
        nop                               ; [0] AGU stall
        suba      r6,sp                    ; [29]
DW_20
    [
        pop       r6                      ; [29]
        pop       r7                      ; [29]
    ]
DW_21
    rts                                                  ; [29]
Ffct_end

bb_cs_offset__fct_1    equ    0            ; at _fct_1 sp = 0
bb_cs_offset_DW_23     equ    2            ; at DW_23 sp = 2
bb_cs_offset_DW_41     equ    26           ; at DW_41 sp = 26
bb_cs_offset_DW_42     equ    24           ; at DW_42 sp = 24

;*****
;
; Function _fct_1
;
; Stack frame size: 56
;
; Calling Convention: Standard
;
; Parameter pt    passed in register r0
; Parameter pt1   passed in register r1
; Parameter Max   passed in stack with offset -12
;

```

Modulo Addressing Example

```
; Returned value  ret_fct_1  passed in register d0
;
;*****
        global  _fct_1
        align   16
_fct_1  type    func
[
    clr      d0                ; [38]
    push     r6                ; [32]
    push     r7                ; [32]
]
DW_23
[
    move.l   (sp-20), r2        ; [40]
    adda     #48, sp, r6        ; [0]
]
[
    tfra     r6, sp            ; [0]
    tfra     r1, r3            ; [41] B1
]
DW_25
[
    tstega   r2                ; [40]
    move.w   #<6, m0           ; [0] B1
]
    bt       <L4               ; [40]
    move.l   (sp-68), d4        ; [0]
[
    move.w   #2, n3             ; [0]
    move.w   #<14, m1           ; [0]
]
[
    bmset    #2048, mctl.l     ; [0]
    move.w   #<0, d0            ; [0]
]
[
    max      d0, d4             ; [0]
    tfra     r1, r11            ; [0]
    tfra     r0, r10            ; [0]
]
[
```

```

    doensh3  d4                      ; [0]
    bmset    #36864,mctl1.l         ; [0]
]
    tfra     r0,r2                  ; [41]
    nop      ; [0]  L_D_3
    loopstart3
L9
[
    move.w    (r3)+n3,d1             ; [41]
    move.w    (r2)+,d0              ; [41]
]
    add      d0,d1,d0               ; [42]
    loopend3
    bmclr     #61440,mctl1.l        ; [0]
    bmclr     #3840,mctl1.l        ; [0]
L4
    move.w    #48,r6                ; [45]
    nop      ; [0]  AGU stall
    suba     r6,sp                  ; [45]
DW_41
[
    pop       r6                    ; [45]
    pop       r7                    ; [45]
]
DW_42
    rts      ; [45]
Ffct_1_end

TextEnd_ce2
    endsec

```

Modulo Addressing Example

Induction-Related Loop Optimizations

- [Loop Detection and Normalization](#)
- [Loop-Invariant Code Motion](#)
- [Scalarization](#)
- [Pointer Promotion](#)
- [Single-loop Induction Process](#)
- [Sequential Accesses and Related Optimizations](#)
- [Cross-loop Induction](#)

Loop Detection and Normalization

- [Detection of hardware-mappable loops](#)
- [Normalization of hardware loops](#)

Detection of hardware-mappable loops

- [Need and scope](#)
- ["For" loops versus "do-while" loops, internal representation](#)
- [Overview](#)

Need and scope

One of the first stage of the optimizer aims to detect “hardware mappable loops”, i.e. loops that could be candidates for a future mapping as actual hardware loops. This occurs when loop bounds and loop count can be computed at compile time, either as a static or dynamic value. When such a loop is detected, its features are stored in an appropriate data structure that contains the definitions of its bounds, steps, loop counter, ... This step performs no actual transformation.

When such a loop is detected, one also memorizes its main features (static loop count, expression of loop bounds, step, iteration instruction, ...).

"For" loops versus "do-while" loops, internal representation

"For" loops are internally represented as "do while" loops. Let us consider the following "for" loop:

```
for (i=3; i<10; i++) {  
    ...  
}
```

Its internal representation is equivalent to the loop below:

```
i=3;  
do {  
    ...  
    i++;  
} while (i<10);
```

Overview

We present different cases of loops that are detected as hardware mappable loops. Examples involve "for" loops, but equivalent "do while" loops are detected as hardware loops as well.

a. Static loop, unit step

The following loop is the simplest case of hardware loop. It corresponds with a static loop with unit step:

```
for (i=3; i<10; i++) {  
    ...  
}
```

The main features are the following:

- iteration instruction is "i++",
- step is "1",
- expression for lower bound is i=3,
- expression for upper bound is i==10 ,
- related static loop count is 7.

b. Static loop, non-unit step

A similar static loop with a step different from 1 is detected as an hardware loop:

```
for(i=3; i<10; i+=4) {  
    ...  
}
```

The main features are the following:

- iteration instruction is "i+=4",
- step is "4",
- expression for lower bound is i=3,
- expression for upper bound is i==10,
- related static loop count is 2.

c. Static loop, decreasing case

The equivalent decreasing case is detected as hardware loop:

```
for(i=10; i>3; i-=4) {  
    ...  
}
```

The main features are the following:

- iteration instruction is "i-=4",
- step is "-4",
- expression for lower bound is i=10,
- expression for upper bound is i==3,
- related static loop count is 2.

d. Dynamic loop, simple case with unit step

Dynamic loops, i.e. loops whose loop count expression involve dynamic variables (variable is loop invariant, but its value is unknown at compile time) can also be detected as hardware loops:

```
for(i=3; i<high_bound; i++) {  
    ...  
}
```

The main features are the following:

Induction-Related Loop Optimizations

Detection of hardware-mappable loops

- iteration instruction is "i++",
- step is "1",
- expression for lower bound is i=3,
- expression for upper bound is i==high_bound,
- related dynamic loop count is expressed as (high_bound-3).

e. Dynamic loop, simple case with non-unit step

Let us now consider a dynamic loop controlled by an iteration instruction involving a non-unit step. Such a loop is detected as hardware loop if the step is:

- either a power of two,
- or equal to 3, 5, or 7, and if the type of the loop index and bounds is either a short integer or an unsigned short integer.

This restriction is due to the complexity of the expression of the related dynamic loop count and possibility to transform expressions involving division as fractional multiplication(s).

For instance the following loop is detected as a hardware loop:

```
for(i=3; i<high_bound; i+=4) {  
    ...  
}
```

The main features are the following:

- iteration instruction is "i+=4",
- step is "4",
- expression for lower bound is i=3,
- expression for upper bound is i==high_bound,
- related dynamic loop count is expressed as (((high_bound-3-1)+4)>>2)=high_bound>>2.

This loop is detected as an hardware loop as well:

```
short int high_bound  
short int i;  
...  
for(i=3; i<high_bound; i+=3) {  
    ...  
}
```

The main features are the following:

- iteration instruction is "i+=3",
- step is "3",
- expression for lower bound is i=3,
- expression for upper bound is i==high_bound,
- related dynamic loop count is expressed as (((high_bound-3-1)+3) / * 10923), where the "/" operation corresponds with a multiplication followed by a shift by 15.

The following loop is not detected as hardware loops by the StarCore compiler:

```
int high_bound;
int i;
...
for(i=3; i<high_bound; i+=7) {
    ...
}
```

Normalization of hardware loops

- [Need](#)
- [Principles of the normalization](#)
- [Scope](#)
- [Illustration](#)

Need

Let us consider the two following loops:

```
for(i=0 ; i<MAX ; i++)
```

and:

```
for(i=0 ; i<MAX*2 ; i+=2)
```

Those loops are both detected as hardware mappable loops. The need for a normalization is highlighted by two different facts:

- none of those loops corresponds with the hardware loop mechanism of the StarCore machine, which is based on:
- a decreasing loop counter,
- a step equal to -1,
- an initial high bound equal to the number of iteration of the loop (either static or dynamic),

- a final low bound equal to 0.
- those two loops have a similar behavior.

However they are perceived as different, because the features extracted during the detection of hardware mappable loops are different. Transformations that require a loop shape recognition and or comparison are thus more complex than they could.

Principles of the normalization

To overcome these drawbacks, one pre-processes loops so as to "normalize" them. After normalization, one wishes to obtain loops with:

- a decreasing loop counter,
- a step equal to -1,
- an initial high bound equal to the number of iteration of the loop (either static or dynamic),
- a final low bound equal to 0.

This shape is much more tractable to address the actual hardware loop mechanism. The possible combinations of patterns to be recognized and transformed is then dramatically reduced. One makes use of the information extracted for detection of hardware-mappable loops to retrieve useful features.

One then modifies the loop so as:

- to set a new temporary to be used as its loop counter,
- to normalize its step,
- to modify its branching instruction(s).

All former (or "C-like") features of the loops are memorized.

Scope

Normalization is applied to any hardware mappable loops, either static or dynamic.

Illustration

Let us consider the initial C loop below:

```
for(i=3; i<10; i++) {  
    ...  
}
```



```
}
```

Its initial internal representation is an equivalent "do-while" loop:

```
i=3;  
do {  
    ...  
    i++;  
} while (i<10);
```

Assuming that normalization is legal for this loop body, its internal representation after normalization process is as follows:

```
lc=7;  
do {  
    ...  
    lc--;  
} while (lc>0);
```

Loop-Invariant Code Motion

Invariant code motion is applied before any transformation of loops and especially before induction process. It consists in moving loop invariant instructions to loop-prolog. The process is performed in each loop. Move is performed from current-loop towards its prolog.

The principle is illustrated by the following C-code:

```
for(i=0; i<10; i++) {  
    for(j=0; j<10; j++) {  
        ...  
        A = i*3;  
        B = j*4;  
        C = 10;  
        array[j] = A+B+C;  
        ...  
    }  
}
```

This code is transformed into:

```
C = 10;  
for(i=0; i<10; i++) {  
    A = i*3;  
    for(j=0; j<10; j++) {  
        ...  
    }  
}
```

```
        B = j*4;  
        array[j] = A+B+C;  
        ...  
    }  
}
```

The instruction "C=10" was moved twice:

- once from inner loop to inner loop prolog,
- once from outer loop to outer loop prolog.

Scalarization

- [Need and scope](#)
- [Overview and goal](#)
- [Assembly view and result](#)

Need and scope

The need for scalarization occurs when a memory location is read and written in a loop, thanks to an invariant pointer or index. In this case the detection can be based on a simple test on the invariance of the variables. The accesses can then be "scalarized", i.e. moved outside loop, and related access to memory content inside loop replaced by scalar operations. This case especially occurs when a memory location is used as an accumulator inside a loop, for instance in FIR applications.

Overview and goal

Let us consider the piece of code below, where *i* is assumed to be invariant in inner loop:

```
for(...) {  
    ...  
    A[i] = 0;  
    for(...) {  
        A[i] = A[i] + f(j);  
        j++;  
    }  
    i++;  
}
```

The principle for the scalarization consists in replacing the accumulation through read/write memory by an accumulation in a scalar. This scalar variable is set and read outside the loop. The equivalent C code is as follows:

```
for(...) {
    ...
    scalar = A[i];
    for(...) {
        scalar = scalar + f(j);
        j++;
    }
    A[i] = scalar;
    i++;
}
```

Assembly view and result

Initial C code

The initial code corresponds with an accumulation, like that one can find in a FIR filter:

```
#define MAX 10
...
short tab[MAX][MAX];
int   accu[MAX];
int   i, j;
...
for(i=0; i<MAX; i++) {
    accu[i]=0;
    for(j=0; j<MAX; j++) {
        accu[i]=accu[i]+tab[i][j];
    }
}
```

Code without scalarization

By default the `accu[i]` array is accessed twice inside the inner-loop body: it is read first and then written. In the code below this read/write access is performed thanks to the register `r0`, in bold font:

Induction-Related Loop Optimizations

Assembly view and result

```
DW_2
    move.w    #240,r6                ;[20]
    nop                      ;[0] AGU stall
    adda      r6,sp                ;[20]
DW_5
    adda      #>-40,sp,r0            ;[27]
    adda      #>-240,sp,r1          ;[0]
    doen2     #<10                  ;[0]
    dosetup2   L10                  ;[0]
    sub       d0,d0,d0              ;[0];CLR instruction
    loopstart2
L10
    move.l     d0,(r0)               ;[27]
    doen3     #<10                  ;[0]
    dosetup3   L9                   ;[0]
    falign
    loopstart3
L9
    move.l     (r0),d1               ;[29]
    move.w     (r1),d2               ;[29]
    add        d1,d2,d3              ;[29]
    move.l     d3,(r0)               ;[29]
    adda      #<2,r1                 ;[28]
    loopend3
L6
    adda      #<4,r0                 ;[26]
    loopend2
```

Code with scalarization

When the compiler detects and simplifies this kind of patterns, it reduces the number of memory accesses and instructions in the inner loop body. The accumulation is performed in a data register instead (d0 in the code below). This register is cleared first in outer loop body. It is read and its content stored in the appropriate memory slot, in outer loop as well:

```
DW_2
    move.w    #240,r6                ;[20]
    nop                      ;[0] AGU stall
    adda      r6,sp                ;[20]
DW_5
```

```

    adda    #>-40,sp,r0          ; [27]
    adda    #>-240,sp,r1         ; [0]
    doen2   #<10                 ; [0]
    dosetup2 L10                 ; [0]
    loopstart2
L10
    move.w   #<0,d0              ; [27]
    move.l   d0,(r0)             ; [27]
    doen3    #<10                 ; [0]
    dosetup3 L9                 ; [0]
    nop      ; [0]   L_D_6
    falign
    loopstart3
L9
    move.w   (r1),d1             ; [29]
    iadd     d1,d0               ; [29]
    adda     #<2,r1              ; [28]
    loopend3
L6
    move.l   d0,(r0)             ; [0]
    adda     #<4,r0              ; [26]
    loopend2

```

Thanks to this optimization, the number of instructions in inner loop body is now 3 instead of 5. The number of memory accesses in inner loop body is now 1 instead of 3.

Pointer Promotion

- [Need and scope](#)
- [Overview and goal](#)
- [Assembly view and result](#)

Need and scope

Indexed memory accesses are known to be less efficient than indirect ones. The compiler thus transforms indexed accesses into indirect ones, especially when they occur inside a loop.

Overview and goal

Let us consider the initial C code below, where array is accessed thanks to an indexed instruction:

```
...
short tab[MAX];
...
for(i=0 ; i<MAX ; i+=step) {
    tab[i] = 0 ;
}...
```

The aim of the pointer promotion consists in making the efficiency of the code independent of the programmer style concerning the access to arrays. In fact the address which is actually used by the machine for the access in the code above is $\&\text{tab} + 2*i$. After optimization the intermediate code is thus equivalent to the following one :

```
...
short tab[MAX];
...
for(i=0 ; i<MAX ; i+=step) {
    *(&tab + 2*i) = 0 ;
}
...
```

Assembly view and result

Initial C code

The initial code represents an indexed access:

```
...
short tab[MAX];
...
for(i=0 ; i<MAX ; i+=step) {
    tab[i] = 0 ;
}
...
```

Code with pointer promotion

The actual assembly code generated (when transformation of induction variables is disabled) corresponds with the equivalent indirect form:

```
...
DW_2
    adda      #<24, sp                ; [20]
DW_3
    sub       d0, d0, d0              ; [27]; CLR instruction
    doen3     #<10                    ; [0]
    dosetup3  L5                      ; [0]
    sub       d0, d0, d1              ; [0]; CLR instruction
    adda      #>-24, sp, r1           ; [0]
    loopstart3
L5
    move.l    d0, r0                  ; [28]
    nop                          ; [0] AGU stall
    asla      r0                      ; [28]
    adda      r1, r0                  ; [28]
    move.w    d1, (r0)                ; [28]
    inc       d0                      ; [27]
    loopend3
...
```

Single-loop Induction Process

- [Introduction](#)
- [Simple induction variables](#)
- [Multi-Step IV](#)
- [Composition of IV](#)
- [Wrap around variables](#)
- [Monotonic variables](#)
- [Modulo-induction](#)
- [Simplification of redundant IV](#)

Introduction

Goal

The aim of the induction process is twofold:

- it performs a strength reduction by moving complex expressions related to induction variables outside the loop body and replacing them by simple additions,
- it also tends to allow a wider and more relevant use of address registers, as soon as induction variables are involved in memory accesses.

Definitions, properties and scope

Only linear induction variables are within the scope of CodeWarrior for StarCore. Linear Induction Variables (IV) are featured by the following properties:

- a basic linear IV is a variable that is either decremented or incremented by an either constant value (literal) or a loop-invariant step within the loop body. An IV which is incremented more than once within the loop-body is said to be a multi-step IV,
- if the step of an IV is a constant (i.e. a literal), then the IV is said to be a static IV. Otherwise if the step is an invariant value in loop, the IV is said to be a dynamic IV,

The key property is that any linear function or linear combination of linear induction variables also defines a linear induction variable:

- a non-basic or derived IV is derived from any other IV by means of a linear function, where the gain is a constant value and the offset is either a constant or a loop-invariant value,
- a composed IV is a linear combination of any other IV, where gains are constant values and the offset is either a constant or a loop-invariant value.

These definitions can be extended to neighboring cases such as wrap-around variables, monotonic variables (with conditional increment in loop), polynomial and geometric induction variables, ... CodeWarrior for StarCore deals with monotonic and wrap-around variables. Polynomial and geometric ones are not within the scope of the current version.

A more systematic classification of IV can be found in [GSW95]. The reader may also refer to key paper concerning induction process and related optimizations [Wol92].

Classification and content

This chapter describes the transformations of induction variables. Three main types of linear IV are concerned:

- simple IV,
- multi-step IV,
- composed IV.

Besides those cases, CodeWarrior for StarCore also takes the following extensions into account:

- modulo IV,
- wrap-around variables,
- monotonic variables.

Remark

The chapters below illustrate the effect of those optimizations. For the sake of simplicity we focus on static steps, i.e. steps known as literal values at compile time. However the principle of the transformation is the same in case of dynamic steps, i.e. steps known to be loop invariant variables, but whose value is unknown at compile time.

Simple induction variables

Overview and goal

Let us consider the initial C code below:

```
...
short tab[MAX];
...
for(i=0 ; i<MAX ; i++) {
    tab[i] = 0 ;
}
...
```

Because of the pointer promotion, the code below is strictly equivalent:

Induction-Related Loop Optimizations

Simple induction variables

```
...
short tab[MAX];
...
for(i=0 ; i<MAX ; i++) {
    *(&tab + 2*i) = 0 ;
}
...
```

Each access to the "tab" array thus requires one extra multiplication and one extra addition.

The "i" variable is a basic induction variable, whose step is equal to "1". As a consequence, the following variables are derived induction variables as well:

- (2*i) is a derived IV with step 2, and initial value 0,
- as the address of the "tab" array is a loop invariant variable, then the result of the (&tab + 2*i) expression is a derived IV too. Its step is equal to 2. Its initial value is &tab.

The induction process aims to bring to the code into an equivalent and more efficient form, where the pointer is set before entering the loop, and incremented inside the loop body. Moreover the intermediate expression (and variable) "2*i" is removed because it is no longer used :

```
...
short tab[MAX];
short* pt = &tab;
...
for(i=0 ; i<MAX ; i++) {
    *pt = 0 ;
    pt = pt + 2;
}
...
```

Assembly view and result

a. Original C source code

Let us consider the following code, which contains an indexed access using an inducted index:

```
...
short tab[MAX];
...
for(i=0 ; i<MAX ; i++) {
```

```
        tab[i] = 0 ;  
    }  
    ...
```

b. Assembly code without induction

Without any transformation of induction variables, the actual address used for memory access is recomputed each time the loop is iterated. The loop body thus contains 5 actual instructions and one nop:

```
    ...  
DW_3  
    sub        d0,d0,d0                ; [26] ; CLR  
instruction  
    doen3      #<10                    ; [0]  
    dosetup3    L5                      ; [0]  
    sub        d0,d0,d1                ; [0] ; CLR  
instruction  
    adda        #>-24,sp,r1             ; [0]  
    loopstart3  
L5  
    move.l      d0,r0                  ; [27]  
    nop                     ; [0] AGU stall  
    asla        r0                     ; [27]  
    adda        r1,r0                  ; [27]  
    move.w      d1,(r0)                ; [27]  
    inc         d0                     ; [26]  
    loopend3  
    ...
```

c. Assembly code with induction

The induction process moves the initialization of the pointer to the loop prolog, and replaces the arithmetic expressions inside the loop body by one single addition. Moreover one notices several improvements which are directly related to the induction process:

- one single address register can be used to perform and monitor the memory access (r0),
- as the i variable is no longer useful, the related induction and initialization instructions are removed.

As a consequence the loop body only contains 2 instructions instead of 6 before:

```
...
DW_3
    adda    #>-24,sp,r0          ; [27]
    doensh3 #<10                 ; [0]
    sub     d0,d0,d0            ; [0] ; CLR
instruction
    loopstart3
L5
    move.w  d0,(r0)              ; [27]
    adda    #<2,r0               ; [26]
    loopend3
...
```

Remark and limitations concerning divisions and right-shift operations

Induction process handles restricted cases of divisions:

- candidate must be a non-basic IV derived from a static basic IV by means of a division by a power of two (i.e. right shift),
- if such a candidate is accepted as non-basic IV, then derived IV are currently not detected.

Other cases of divisions are not handled by the current version of the compiler.

Multi-Step IV

Overview and goal

Let us consider the initial C code below:

```
...
short tab1[MAX];
short tab2[MAX];
...
for(i=0 ; i<MAX ; ) {
    tab1[i] = 0 ;
    i++;
    tab2[i] = 1;
    i++;
}
```

...

Here the "i" variable is inducted twice inside the loop body. It is said to be a multi-step induction variable. Because of the pointer promotion, the code below is strictly equivalent:

```
...
short tab1[MAX];
short tab2[MAX];
...
for(i=0 ; i<MAX ; ) {
    *(&tab1 + 2*i) = 0 ;
    i++;
    *(&tab2 + 2*i) = 1 ;
    i++;
}
...
```

Derived IV are detected for each value of the multi-step basic IV:

- the expression $(&\text{tab1} + 2*i)$ defines derived induction variables for the initial value of i,
- the expression $(&\text{tab2} + 2*i)$ defines derived induction variables for the first step value of i (i.e. $i+1$).

At this stage, several strategies can be chosen to transform multi-step IV. By default, CodeWarrior for StarCore de-correlates all the derived IV. This tends to increase the need for register, but correlatively reduces false data dependencies and increases the fine grain parallelism of the code. We shall see later that one sometimes makes different choices. The induction process brings the code to an equivalent and more efficient form:

```
...
short tab1[MAX];
short tab2[MAX];
short* pt1 = &tab1;
short* pt2 = &tab2 + 2
...
for(i=0 ; i<MAX ; i+=2) {
    *pt1 = 0 ;
    *pt2 = 1;
    pt1 = pt1 + 4;
}
```

```

        pt2 = pt2 + 4;
    }
    ...

```

Assembly view and result***a. Original C source code***

The loop now contains two indexed accesses. The index is inducted twice:

```

...
short tab1[MAX];
short tab2[MAX];
...
for(i=0 ; i<MAX ; ) {
    tab1[i] = 0 ;
    i++;
    tab2[i] = 1;
    i++;
}
...

```

b. Assembly code without multi-step induction

The code obtained without any transformation of multi-step IV is dumped below. The loop body contains 13 instructions (3 nop instructions):

```

...
DW_2
    move.w    #40,r6                ;[7]
    nop                          ;[0] AGU stall
    adda      r6,sp                 ;[7]
DW_5
    sub       d0,d0,d0             ;[15];CLR
instruction
    doen3     #<5                  ;[0]
    dosetup3  L5                   ;[0]
    move.w    #<1,d1               ;[0]
    adda      #>-20,sp,r1          ;[0]
    adda      #>-40,sp,r3          ;[0]
    loopstart3

```

```

L5
    move.l    d0,r0                ;[16]
    nop                      ;[0] AGU stall
    asla      r0                ;[16]
    adda      r1,r0              ;[16]
    sub       d0,d0,d2           ;[16];CLR
instruction
    move.w    d2,(r0)             ;[16]
    inc       d0                 ;[17]
    move.l    d0,r2              ;[18]
    nop                      ;[0] AGU stall
    asla      r2                ;[18]
    adda      r3,r2              ;[18]
    move.w    d1,(r2)            ;[18]
    inc       d0                 ;[19]
    loopend3
...

```

c. Assembly code with multi-step induction

If multi-step IV are detected and transformed, then the loop body only contains 4 instructions. One address register is used to access each array tab1 and tab2. Those registers are inducted once each:

```

...
DW_2
    move.w    #40,r6              ;[7]
    nop                      ;[0] AGU stall
    adda      r6,sp              ;[7]
DW_5
    adda      #>-38,sp,r0          ;[18]
    adda      #>-20,sp,r1          ;[16]
    doen3     #<5                 ;[0]
    dosetup3  L5                  ;[0]
    sub       d0,d0,d0           ;[0];CLR
instruction
    move.w    #<1,d1              ;[0]
    loopstart3
L5
    move.w    d0,(r1)             ;[16]
    move.w    d1,(r0)             ;[18]
    adda      #<4,r1              ;[19]
    adda      #<4,r0              ;[19]

```

```
        loopend3  
    ...
```

Composition of IV

Overview and goal

Let's now consider the initial C code, corresponding with a diagonal access to matrix:

```
    ...  
    short tab[10][20];  
    ...  
    for(i=0, j=0 ; i<10 ; ) {  
        tab[i][j] = i ;  
        i++ ;  
        j+=2 ;  
    }  
    ...
```

Because of the pointer promotion, the code below is strictly equivalent:

```
    ...  
    short tab[10][20];  
    ...  
    for(i=0, j=0 ; i<10 ; ) {  
        *(&tab1 + 2*(20*i+j)) = i ;  
        i++ ;  
        j+=2 ;  
    }  
    ...
```

Here "i" and "j" are both induction variables with different steps. The result of the linear combination $(20*i+j)$ defines an induction variable. Its step can be computed using the same linear combination: $\text{step} = 20*1 + 2 = 22$. The result of the multiplication of this expression by 2 is also an IV whose step is $2*22=44$.

The induction process thus brings the code to an equivalent and more efficient form:


```
...
short tab[10][20];
short *pt = &tab[0][0];
...
for(i=0, j=0 ; i<10 ; ) {
    *pt = i ;
    i++ ;
    pt=pt+44;
}
...
```

Assembly view and result

a. Original C source code

The indexed access is based on a linear combination of "i" and "j", which are both loop induction variables:

```
...
short tab[10][20];
...
for(i=0, j=0 ; i<10 ; ) {
    tab[i][j] = i ;
    i++ ;
    j+=2 ;
}
...
```

b. Assembly code without composed induction

Each of the two components are identified as loop induction variables: $(40*i)$ is stored in register d2, whereas $(4*i)$ is stored in register d1. The sum of these two induction variables is not detected as an induction variable. As a consequence, the loop body contains 8 instructions and the code is as follows:

```
...
DW_2
    move.w    #400, r6                ; [7]
    nop                          ; [0] AGU stall
    adda      r6, sp                ; [7]
DW_5
```

Induction-Related Loop Optimizations

Composition of IV

```
        sub      d0,d0,d0                ; [14] ;CLR
instruction
        sub      d0,d0,d1                ; [15] ;CLR
instruction
        sub      d0,d0,d2                ; [15] ;CLR
instruction
        doen3    #<10                    ; [0]
        dosetup3 L5                      ; [0]
        adda     #>-400,sp,r1             ; [0]
        loopstart3
L5
        add      d1,d2,d3                ; [15]
        move.l   d3,r0                   ; [15]
        nop      ; [0] AGU stall
        adda     r1,r0                   ; [15]
        move.w   d0,(r0)                 ; [15]
        add      #<4,d1                  ; [17]
        inc      d0                      ; [16]
        addnc.w  #40,d2,d2               ; [16]
        loopend3
L4
...

```

c. Assembly code with composed induction

With the transformation of composed IV, the linear combination of *i*, *j* and base offset *tab*, is detected as a new induction variable. The code is optimized as follows, and the loop body only contains 3 instructions:

```
...
DW_2
        move.w   #400,r6                 ; [7]
        nop      ; [0] AGU stall
        adda     r6,sp                   ; [7]
DW_5
        sub      d0,d0,d0                ; [14] ;CLR
instruction
        adda     #>-400,sp,r0             ; [0]
        doen3    #<10                    ; [0]
        dosetup3 L5                      ; [0]
        nop      ; [0] L_D_6
        loopstart3

```

```
L5
    move.w    d0, (r0)                ; [15]
    inc       d0                      ; [16]
    adda      #>44, r0, r0            ; [16]
    loopend3
L4
...
```

Wrap around variables

Overview and goal

Wrap around variables corresponds to variables that follow an induction law only after the first iteration of the loop.

A dummy example of wrap around variable is provided below:

```
...
wrap = f(...);
...
i=0;
for(...) {
    ...array[wrap] ...;
    wrap = i;
    i++;
}
...
```

The "wrap" variable is a wrap-around variable: its value depends on that of the "i" basic IV, except during the first iteration of the loop.

Such a variable cannot be transformed using the standard induction scheme. The solution consists in first "peeling" the loop once and then applying usual induction process. Loop peeling consists in moving one iteration of the loop outside the body. Thus, transforming wrap-around variables is twofold:

- during the detection step of induction process:
- one detects patterns corresponding to wrap-around-variables,
- as soon as such a pattern is detected, one checks if loop can be peeled once,

Induction-Related Loop Optimizations

Wrap around variables

- if yes, then variable is memorized in the temporary data structure of IV, and the need for loop peeling is memorized too,
- before the transformation step of induction process, loop is actually peeled once, so that wrap around variables can then be processed like any other IV.

Loop peeling leads to the following code that do not contain any actual wrap around variable (the two definitions reaching usage of "wrap" in loop are now equivalent and a usual induction process can now be applied):

```
...
wrap = f(...);
...
i=0;
/** first iteration peeled **/
...array[wrap]...;
wrap = i;
i++;
/*****/
for(...) {
    ...array[wrap]...;
    wrap = i;
    i++;
}
...
```

In fact the code is now equivalent to the following one:

```
...
wrap = f(...);
...
i=0;
...array[wrap]...;
i=1;
for() {
    wrap = i-1;
    ...array[wrap]...;
    i++;
}
...
```

Assembly view and result

As soon as the loop has been peeled, then the transformation is identical to that performed for regular IV. The reader may then refer to the previous chapters for the assembly view of such a transformation.

Monotonic variables

Overview and goal

Monotonic variables are special IV whose induction instruction is executed under a given condition:

```
...
for(...) {
    ...
    non_basic_iv = basic_iv * 3;
    if(...) {
        basic_iv=basic_iv+1;
    }
}
...
```

The CodeWarrior for StarCore handles those variables. The related non-basic IV inherit this property, i.e. also have a conditional induction:

```
...
for(...) {
    ...
    if(...) {
        basic_iv=basic_iv+1;
        non_basic_iv = non_basic_iv + 3;
    }
}
...
```

Assembly view and result

There is no other fundamental difference with non-conditional IV. Thus the assembly code is similar.

Modulo-induction

Overview and goal

Detection and transformation of IV described in the previous sections only concerned pure linear IV. In many cases, signal processing applications make use of operations such as modulo and “bit-reverse” (which is roughly speaking a “reversed” counter where MSB is incremented first and carry propagated from MSB towards LSB). Even if these operations are no longer pure linear ones, they can (and should!) be part of the induction process. Otherwise the modulo-addressing capabilities of the processor cannot be efficiently addressed.

Let us consider the following source code:

```
...
short tab[MAX];
...
for(i=0, j=0 ; j<MAX ; j++) {
    tab[i] = 0 ;
    i += 1;
    i = i % 8;
}
...
```

Due to the pointer promotion, the code can be rewritten as follows:

```
...
short tab[MAX];
...
for(i=0, j=0 ; i<MAX ; j++) {
    *(&tab + 2*i) = 0 ;
    i=i+1;
    i=i%8;
}
...
```

In this code, the “i” variable is a basic modulo IV, i.e. an IV which is both incremented and modified by a modulo operation. Subsequently, the following expressions also have a modulo inducted behavior:

- the $(2*i)$ multiplication is a derived modulo IV, whose step is 2 and modulo operator is $8*2=16$,
- the $(\&\text{tab} + (2*i))$ addition has a more complex behavior:
- it behaves like an IV with step 2,
- its initial value is $\&\text{tab}$,
- its value is reset to $\&\text{tab}$ each time it reaches a threshold equals $(\&\text{tab} + 16)$.

In other words, an equivalent form of the code can be obtained:

```
...
short tab[MAX];
short* pt=&tab[0];
...
for(i=0, j=0 ; i<MAX ; j++) {
    *pt = 0 ;

    pt=pt+2;
    pt=pt-&tab;
    pt=pt%16;
    pt=pt+&tab;
}
...
```

This form corresponds with a modulo-addressing patterns applied to "pt" pointer, with base equal to $\&\text{tab}$, and modulo operator equal to 16.

Assembly view and result

a. Initial C code

A modulo access is performed using "i" as an index:

```
...
short tab[MAX];
...
for(i=0, j=0 ; j<MAX ; j++) {
    tab[i] = 0 ;
    i += 1;
    i = i % 8;
}
...
```

b. Assembly code without a modulo induction

In any case the compiler replaces modulo operator by a conditional subtraction as soon as it is legal to do so (sequence in bold font). Otherwise the appropriate runtime is called (Qmod32, Qmod16), resulting in a less efficient code.

```

...
DW_2
    adda      #<24, sp                ; [7]
DW_3
    adda      #>-24, sp, r0           ; [15]
    doen3     #<10                    ; [0]
    dosetup3  L5                      ; [0]
    sub       d0, d0, d0              ; [0] ; CLR
instruction
    move.l    r0, d1                  ; [0]
    adda      #>-24, sp, r1           ; [0]
    move.l    r1, d2                  ; [0]
    loopstart3
L5
    move.l    d1, r0                  ; [15]
    nop                          ; [0] AGU stall
    move.w    d0, (r0)                ; [15]
    adda      #<2, r0                 ; [16]
    move.l    r0, d1                  ; [16]
    sub       d2, d1, d3              ; [16]
    cmpgt.w   #<15, d3                ; [16]
    jf        L7                      ; [16]
    sub       #<16, d1                ; [16]
L7
    nop                          ; [0] L_L_1
    nop                          ; [0] L_L_1
    nop                          ; [0] L_C_1
    loopend3
L4
...

```


c. Assembly code with modulo induction

Thanks to the detection and transformation of the modulo induction, one obtains a compact code that makes use of the StarCore modulo-addressing mode. One notice the `bmset` and `bmclr` instructions which control the modulo addressing mode. Here one makes use of the address register `r0`:

```
...
DW_3
    adda      #>-24,sp,r0          ; [15]
    move.w    #<16,m0             ; [0]
    bmset     #8,mctl.l           ; [0]
    doen3     #<10                ; [0]
    dosetup3   L5                 ; [0]
    sub       d0,d0,d0            ; [0] ; CLR
instruction
    adda      #>-24,sp,r8          ; [0]
    loopstart3
L5
    move.w     d0,(r0)             ; [15]
    nop                          ; [16]
    adda       #<2,r0              ; [16]
    loopend3
L4
    bmclr      #15,mctl.l          ; [0]
...
```

Remarks, cautions and limitations

a. Different classes of modulo IV

a.1 Basic modulo-IV and derivation through multiplication The example above corresponds with:

- a basic modulo IV, i.e. an variable which is both incremented and modified by a modulo operator,
- non-basic IV derived from this basic modulo IV by means of a multiplication.

However other kinds of modulo IV are taken into account by the ENTERRPISE compiler. The corresponding patterns are described below.

a.2 Non-basic modulo IV A modulo IV can be created if a variable is derived from a basic (non-modulo) IV by means of a modulo operator. This corresponds with the case below:

```
...
i = 0 ;
j = 0 ;
for(j=0 ; j<10 ; j++) {
    tab[i%8] = 0 ;
    i += 1;
}
...
```

a.3 Multi-step modulo IV In some cases, applications include some sequence of modulo induction. This may be done by means of:

- either a basic IV which is both a modulo and a multi-step IV (sequence of increment / modulo instruction pairs) or
- non-basic modulo IV derived from a standard multi-step IV.

The first case corresponds with the code below:

```
...
i = 0 ;
j = 0 ;
for(j=0 ; j<60 ; j++) {
    tab[i] = 0 ;
    i += 1;
    i = i % 8;
    tab[i] = 0 ;
    i += 1;
    i = i % 8;
    tab[i] = 0 ;
    i += 1;
    i = i % 8;
}
...
```

The second case is dumped below:

```
...
i = 0 ;
j = 0 ;
for(j=0 ; j<60 ; j++) {
```

```
        tab[i%8] = 0 ;  
        i += 1;  
        tab[i%8] = 0 ;  
        i += 1;  
        tab[i%8] = 0 ;  
        i += 1;  
    }  
    ...
```

b. Limitations

b.1 Non-basic IV derived from modulo basic IV through addition Contrary to other IV, non-basic IV derived from modulo basic IV through addition are not currently transformed (except concerning the addition of a base address with an inducted offset). Namely, the modulo operation can be applied to this non-basic IV iff either the constant or invariant which is added to the basic IV is subtracted prior to the modulo operation and added right after it. The gain to be expected is rather poor.

b.2 Non-basic IV through successive modulo operations

If an IV is derived by means of modulo instruction from an IV that is either a modulo basic IV or a modulo non-basic IV, then it is not recognized as an IV.

b.3 Composition of modulo IV As a consequence of restriction mentioned in b.1, modulo IV cannot currently be composed with any other IV (either basic or non-basic).

c. Remark concerning the conditional subtraction

On some machines like the StarCore, the actual “ modulo ” operation cannot be mapped directly, and jumping to the corresponding subroutine is not very efficient. On the other side, the conditional subtraction is often available as a micro-instruction. In this case, and if some other conditions are verified, modulo instruction is profitably replaced by the conditional subtraction.

Namely, the conditional subtraction:

```
var = var csub cst
```

is equivalent to the following piece of code:

```
if (var>=cst) var = var - cst
```

It is equivalent with the modulo instruction:

```
var = var % cst
```

iff the input value (var in formulas above) is within the appropriate range, i.e. $[0, 2 \cdot \text{cst}]$. In order to replace a modulo operation by an equivalent conditional subtraction, one must check that this condition is always satisfied. This substitution becomes possible and is carried out iff:

- the induction variable (var) is incremented (i.e. actual step is positive) and
- the initial value of this variable can be computed at compile time (static IV) and
- it is not out of the $[0, \text{cst}]$ range.

Otherwise, modulo operation is left unchanged.

d. Accessing the value of a modulo pointer/array after loop

In real-life application, the actual value of a modulo pointer/array may be reused after loop.

d.1 Inefficient C code The most natural way to write this is as follows (case of a modulo array – the sequence of increment/modulo operations is often replaced by a macro):

```
...
int* fct_2()
{
#pragma noline

    int  j          = 0;
    int  l          = 0;
    int* pt         = 0;

    l=0;
    for (j=0; j<5; j++) {
        tab[l]=j;
        l++; l=l%3;
    }

    return(&tab[l]);
}
...
```

For various reasons (the "l" modulo index is reused outside loop, as well as the actual modulo pointer), this pattern leads to pretty inefficient code:

```

...
_fct_2    type    func OPT_SPEED
[
    sub      d0,d0,d0                      ; [32] ;CLR
instruction
    sub      d0,d0,d3                      ; [33] ;CLR
instruction
    doen3    #<5                          ; [0]
    dosetup3 L25                          ; [0]
]
[
    move.l   #_tab,r0                      ; [34]
    move.w   #<3,d4                        ; [0]
]
[
    bmset    #8,mctl.l                     ; [35]
    move.w   #<12,m0                       ; [35]
]
    nop                                ; [0] A_1
    move.l   #_tab,r8                      ; [0]
    falign
    loopstart3
L25
[
    inc      d0                            ; [35]
    inc      d3                            ; [36]
    move.l   d3,(r0)+                      ; [35]
]
    cmpgt    d0,d4                        ; [35]
    iff sub  d4,d0,d0                      ; [35]
    loopend3
[
    bmclr    #15,mctl.l                    ; [0]
    move.l   d0,r0                        ; [38]
]
    nop                                ; [0] AGU stall
[
    asl2a    r0                            ; [38]

```

```
        rtsd                                ; [38]
    ]
        adda    r8,r0                        ; [38]

        global  F_fct_2_end
F_fct_2_end
...

```

d.2 A better solution A very simple modification consists in avoiding actual modification of main induction variable (i.e. "l") by means of the modulo operator. The index used to access array is simply derived from l. The address to be returned is derived from "l" the same way:

```
...
int* fct_2()
{
    #pragma noline

        int  j            = 0;
        int  l            = 0;
        int* pt           = 0;

        l=0;
        for (j=0; j<5; j++) {
            tab[l%3]=j;
            l++;
        }

        return(&tab[l%3]);
}
...

```

Thanks to this simple modification one now get the code below, saving both code size and cycles:

```
...
        global  _fct_2
        align   16
_fct_2    type   func OPT_SPEED
[

```

```

        sub      d0,d0,d4                      ; [33] ;CLR
instruction
        doensh3  #<5                          ; [0]
    ]
        move.l   #_tab,r0                      ; [34]
    [
        bmset    #8,mctl.l                     ; [35]
        move.w   #<12,m0                      ; [35]
    ]
        nop                                           ; [0]A_1
        move.l   #_tab,r8                      ; [0]
        loopstart3
    [
        inc      d4                             ; [36]
        move.l   d4,(r0)+                      ; [35]
    ]
        loopend3
    [
        bmclr    #15,mctl.l                    ; [0]
        rtsd                                           ; [38]
    ]
        move.l   #_tab+8,r0                    ; [38]

        global   F_fct_2_end
F_fct_2_end
...

```

Simplification of redundant IV

Need and scope

Two IV are said to be redundant if they basically obey to the same induction law and can be substituted to each other. More precisely:

- basic IV are redundant if:
- their initial values before entering the loop are the same (or, in some cases, only differ by a constant (literal) value),
- they evolve in the same way at each iteration of the loop, i.e. if they have similar steps and "induction footprint" (same relative location of induction and uses, especially for multi-step IV);
- non-basic IV are redundant if:

- they are derived from the same basic IV or from redundant basic IV,
- making use of the same linear features (gain and offset).

As redundant IV may appear both in the initial object code and after some steps of induction process, detecting and replacing redundant IV is an essential feature. It leads to a more efficient code and reduces code size and register pressure.

Overview and goal

a. Case of redundant basic IV

Redundant basic IV are detected comparing initialization instructions and steps. When two basic IV are found to be redundant, then the two sets of related non-basic IV are merged and related instructions modified. In the code below, *i* and *j* are redundant basic IV, which are composed:

```
...
for (i=0, j=0; square(i+j) <= 25; i++, j++) {
    k *= k;
}
...
```

The code can be transformed so as to be equivalent to the code below:

```
...
for (i=0; square(2*i) <= 25; i++) {
    k *= k;
}
...
```

b. Case of redundant non-basic IV:

Redundant non-basic IV are detected comparing their expressions as linear combination of basic IV. When two non-basic IV are found to be redundant, then one is removed and replaced by the other one. In the dummy code below the two successive expressions derived from *i* are equivalent and redundant:

```
...
short tab[10][20];
```



```
...
for(i=0; i<10 ; i+=2) {
    ...use(2*(20*i)+4) ...;
    ...
    ...use(4*((10*i)+1)) ...;
    ...
}
...
```

However without any detection of redundant non-basic IV they would be considered as different and lead to the computation of two IV. This step is also important because detecting equivalent linear expressions may be useful during the extraction of sequential memory accesses, that is described below. The simplification of redundant non-basic IV leads to the equivalent form below:

```
...
short tab[10][20];
...
for(i=0, j=0 ; i<10 ; i+=2) {
    ...use(2*(20*i)+4) ...;
    ...
    ...use(2*(20*i)+4) ...;
    ...
}
...
```

Assembly view and result

a. Initial C code

In the code below one notices two redundant expressions derived from the "i" basic IV:

```
...
short tab[10];
int i;

for(i=0; i<10 ; i+=2) {
    tab[i]=(2*(20*i)+4);
    tab[i+1]=(4*((10*i)+1));
}
...
```

b. Assembly code without any simplification

If those redundant non-basic IV are neither detected nor simplified, then two separate registers are used to set and update the corresponding values (\$d1 and \$d2). One also notices the compiler detected the opportunity for a double move, which explains the need for 3 registers instead of 2:

```

...
DW3
    move.w    #<4,d1                ; [16]
    move.w    #<4,d2                ; [15]
    adda      #>-24,sp,r0           ; [15]
    doenc3    #<5                   ; [0]
    dosetup3  L5                   ; [0]
    loopstart3
L5
    tfr       d1,d3                ; [15]
    move.2w   d2:d3,(r0)           ; [15]
    adda      #<4,r0                ; [14]
    addnc.w   #80,d2,d2            ; [14]
    addnc.w   #80,d1,d1            ; [14]
    loopend3
L4
...

```

c. Assembly code with simplification

Thanks to the detection of the redundant IV, the number of registers used and instructions in the loop body is reduced as follows. The double move was activated as well:

```

...
DW_3
    move.w    #<4,d0                ; [16]
    adda      #>-24,sp,r0           ; [15]
    doenc3    #<5                   ; [0]
    dosetup3  L5                   ; [0]
    loopstart3
L5
    tfr       d0,d1                ; [15]
    move.2w   d0:d1,(r0)           ; [15]
    adda      #<4,r0                ; [14]

```

```
addnc.w    #80,d0,d0          ; [14]
loopend3
...
```

Sequential Accesses and Related Optimizations

- [Introduction](#)
- [Basic transformation of sequential accesses, control strategy](#)
- [Simplification of redundant memory accesses](#)
- [Access packing \(vectorization\)](#)

Introduction

Definition and scope

An important step of the loop optimizer concerns the detection and some transformations of the so-called sequences of memory accesses or sequential accesses.

Sequences of memory accesses are defined as sets of memory accesses arising in either a loop or a loop nest, such as:

- they concern the same array or base pointer,
- for the sake of code size and efficiency, the distance (or "stride") between two successive memory accesses must be computable as a literal value at compile time (from a theoretical point of view, this condition is not a necessary one). In other words the related IV must be linearly derived from the same basic IV (or from redundant ones), thanks to the same linear gain, and different literal linear offset,
- the pointer must not be redefined between the two accesses, especially through aliases.

Sequences are structured in order to reflect the control flow graph, like in the figure below. Sequences based on modulo IV are also detected.

Figure 1: illustration of the sequence structure (values in the circles represent the strides)

Main interests and goals

Sequences are powerful tools to optimize loops and detect good candidates for several smart loop transformations. The main ones are the following:

- monitoring multi-step IV transformations in order to find a good tradeoff between register pressure / code mobility / code size,
- detection of candidates for multiple moves (so called "access packing"),
- simplification of redundant memory accesses,
- refinement of data dependency analysis,
- software pipelining,
- ...

Basic transformation of sequential accesses, control strategy

Principle

In the current version of the compiler, the transformation consists in converting the sequences from the initial scheme with multiple pointers/address registers, according to the optimization criterion (either speed or size).

Let us consider the case of a sequence based on a multi-step IV. A neighboring code was already presented in the paragraph upon multi-step IV. The main difference is that one now accesses to different slots of the same array:

```
...
short tab[MAX];
...
for(i=0 ; i<MAX ; ) {
    tab[i] = 0 ;
    i++;
    tab[i] = 1;
    i++;
}
...
```

We know that this code can be rewritten as follows:

```
...
```

```
short tab[MAX];
...
for(i=0 ; i<MAX ; ) {
    *(&tab + 2*i) = 0 ;
    i++;
    *(&tab + 2*i) = 1 ;
    i++;
}
...
```

a. Two possible schemes

One can first make use of independent variables for each different value of the mother IV:

```
...
short tab[MAX];
short* pt1 = &tab;
short* pt2 = &tab + 2;
...
for(i=0 ; i<MAX ; i+=2) {
    *pt1 = 0 ;
    *pt2 = 1;
    pt1 = pt1 + 4;
    pt2 = pt2 + 4;
}
...
```

One can also use a single pointer:

```
...
short tab[MAX];
short* pt = &tab;
...
for(i=0 ; i<MAX ; i+=2) {
    *pt = 0 ;
    pt = pt + 2;
    *pt = 1;
    pt = pt + 2;
}
...
```

b. Control strategy

The assembly translations of each of those two forms are not equivalent:

- the first one increases the register pressure, but it also improves the code mobility (i.e. the opportunities for parallelism). Thus it will be chosen as soon as code is optimized for speed, provided that the estimate of the register pressure is low enough to avoid spills;
- the second one results in a poor code mobility, but it also reduces register pressure. Thus it will be chosen as soon as the code is optimized for size.

Assembly view**a. Initial source code**

Let us start from a similar source code, involving one single sequence (for illustration purpose, we slightly modified the strides to avoid multiple moves):

```
...
short tab[MAX];
...
for(i=0 ; i<MAX ; ) {
    tab[i] = 0 ;
    i++;
    tab[i] = 1;
    i+=2;
}
...
```

b. Code optimized for speed

The first version is obtained using the default behavior for multi-step IV when code is optimized for speed. Two address registers (\$r0 and \$r1) are used to perform the related accesses:

```
...
DW_3
    adda      #>-22, sp, r0          ; [19]
    adda      #>-24, sp, r1          ; [17]
    doen3     #<4                    ; [0]
    dosetup3   L5                     ; [0]
```

```

        sub      d0,d0,d0                ; [0] ; CLR
instruction
        move.w   #<1,d1                ; [0]
        loopstart3
L5
        move.w   d0,(r1)                ; [17]
        move.w   d1,(r0)                ; [19]
        adda     #<6,r1                 ; [20]
        adda     #<6,r0                 ; [20]
        loopend3
...

```

If one now pay a look to assembly when optimization level 3 is chosen, one notices that each loop iteration is performed in one cycle:

```

...
DW3
[
    move.w   #3,n3                    ; [0]
    adda     #>-22,sp,r1              ; [17]
]
DW4
[
    adda     #>-24,sp,r4              ; [15]
    move.w   #<1,d2                  ; [0]
]
    loopstart3
L5
DW5
[
    move.w   d3,(r4)+n3              ; [15]
    move.w   d2,(r1)+n3              ; [18]
]
    loopend3
...

```

c. Code with transformed sequences

On the contrary when code is optimized for size, then only register \$r0 is used:

```
...
```

Induction-Related Loop Optimizations

Basic transformation of sequential accesses, control strategy

```
DW_3
    adda      #>-24,sp,r0          ; [17]
    doen3     #<4                  ; [0]
    dosetup3  L5                  ; [0]
    move.w    #<1,d0              ; [0]
    loopstart3
L5
    sub       d0,d0,d1            ; [17] ;CLR
instruction
    move.w    d1,(r0)             ; [17]
    adda      #<2,r0              ; [18]
    move.w    d0,(r0)             ; [19]
    adda      #<4,r0              ; [20]
    loopend3
...
```

With optimization level 3 and optimization for size, one notices that the iteration of the loop now requires two cycles, but one address register is saved, as well as the related initialization instruction:

```
...
DW_3
    adda      #>-24,sp,r0          ; [17]
    move.w    #2,n3               ; [0]
    move.w    #<1,d1              ; [19]
    loopstart3
L5
    move.w    d2,(r0)+            ; [18]
    move.w    d1,(r0)+n3          ; [20]
    loopend3
...
```

A bestiary

In order to process most usual applications in a relevant way, this mechanisms must take as many cases into account as possible: sequences can be derived from simple IV, multi-step IV, composed IV... This paragraph illustrates different cases that are currently handled by the compiler. All the examples are based on indexed access mode. However equivalent programs based on indirect

addressing mode (pointers) are handled the same way by the optimizer.

a. Basic cases

a.1 Sequence derived from a multi-step IV ...

```
for(...) {
    ...array[i] ...;
    i++;
    ...array[i] ...;
    i++;
    ...array[i] ...;
    i++;
}
```

...

a.2 Sequence derived from a simple step IV with different static offsets ...

```
for(...) {
    ...array[i] ...;
    ...array[i+1] ...;
    ...array[i+2] ...;
    i+=3;
}
```

...

b. Cases of multi-dimensional accesses

b.1 Row-major accesses b.1.1 Derived from a multi-step IV ...

```
for(...) {
    for(...) {
        ...array[i][j] ...;
        j++;
        ...array[i][j] ...;
        j++;
        ...array[i][j] ...;
        j++;
    }
    i++;
}
```

...

b.1.2 Derived from a simple IV with different static offsets

```
...
for(...) {
    for(...) {
        ...array[i][j] ...;
        ...array[i][j+1] ...;
        ...array[i][j+2] ...;
        j+=3;
    }
    i++;
}
...
```

b.2 Diagonal access with composed IV ...

```
for(...) {
    ...array[i][j] ...;
    ...array[i+1][j+1] ...;
    ...array[i+2][j+2] ...;
    j+=3;
    i+=3;
}
...
```

b.3 Partial column-major access : ...

```
for(...) {
    for(...) {
        ...array[i][j] ...;
        ...array[i+1][j] ...;
        ...array[i+2][j] ...;
        j++;
    }
    i+=3;
}
...
```

c. Case of fields in data structures

c.1 Arrays in structures The first case concerns sequential accesses to arrays located in structures. Let us consider the code below. One defines the structured type (Struct_tab). The data structure contains an array of integer (tab):

```
...
typedef struct Type_struct_tab *Pt_struct_tab;
typedef struct Type_struct_tab {
    char    letter;
    int     tab[MAX];
    int     number;
} Struct_tab;
...
```

The successive accesses to the content of the array is now detected as a sequential memory access:

```
...
Struct_tab array_tab;
...
for(ind1=0 ; ind1<MAX/2; ind1+=2) {
    array_tab.tab[ind1]=ind1;
    array_tab.tab[ind1+1]=ind1+1;
}
...
```

c.2 Simple fields in structures The second case concerns sequential accesses to successive fields in structures. We refer to the same structured type:

```
...
typedef struct Type_struct_tab *Pt_struct_tab;
typedef struct Type_struct_tab {
    char    letter;
    int     tab[MAX];
    int     number;
} Struct_tab;
...
```

The accesses to the fields of the array is now detected as a sequential memory access:

```
...
Struct_tab array[10];
...
for(ind1=0 ; ind1<MAX/2; ind1+=2) {
    array[ind1].letter='f';
    array[ind1].number=ind1;
}
...
```

Simplification of redundant memory accesses

Definition and scope

Many programs present redundant memory accesses. This is encountered especially in the case of the scalarization presented above. But other cases may occur. Unfortunately, it is usually very difficult to prove:

- that two memory accesses are made at the same address, and also
- that no interleaved access has modified either the content of the memory or the pointer itself.

Thus, simplifying redundant memory access is a tough job if no further assumption is made. However the sequential memory accesses can be used to perform such a task in a restricted scope. Thanks to this framework, one can prove that two memory accesses are redundant and can be simplified.

Overview and goal

a. Write and read

The first case concerns a coupled write and read memory access. In this case a memory slot is first written, and then read:

```
...
for() {
    ...
    A[i]=X;
    i++;
    A[i]=Y;
    i--;
    Z=A[i];
}
```

```
    ...  
    i++;  
}  
...
```

The two instructions in bold font concern the same memory location. No actual modification occurs between the write access and the related read access. Thus those instructions can be modified so as to reduce the number of memory accesses inside the loop body:

```
    ...  
    for() {  
        ...  
        A[i]=X;  
        i++;  
        A[i]=Y;  
        i--;  
        Z=X;  
        ...  
        i++;  
    }  
    ...
```

b. Double read

The same kind of redundancy may occurs in case of multiple memory read:

```
    ...  
    for() {  
        ...  
        X=A[i] ;  
        i++;  
        A[i]=Y;  
        i--;  
        Z=A[i] ;  
        ...  
        i++;  
    }  
    ...
```

Induction-Related Loop Optimizations

Simplification of redundant memory accesses

In this case the simplification leads to the equivalent code below:

```
...
for() {
    ...
    X=A[i];
    i++;
    A[i]=Y;
    i--;
    Z=X;
    ...
    i++;
}
...
```

Assembly view and result

a. Initial C code

```
...
int tab[MAX];
int i,j;

for(i=0, j=0; j<MAX; j++) {
    tab[i]=j; /*write access at slot i*/
    ...
    tab[i+1]=1;
    ...
    c=tab[i]; /*redundant read access at the
same slot*/
    i++;
}
...
```

b. Assembly code without simplification of redundant accesses

If the optimizer does not take care of the write/read redundant memory access, then this access is performed twice: one for write access, one for read and assignment to c:

```
...
DW_5
    sub        d0,d0,d0                ; [14] ;CLR
instruction
```

```

        adda      #>-36,sp,r0          ; [17]
        adda      #>-40,sp,r1          ; [15]
        doen3     #<10                 ; [0]
        dosetup3  L5                   ; [0]
        move.w    #<1,d1               ; [0]
        loopstart3
L5
        move.l     d0,(r1)              ; [15]
        move.l     d1,(r0)              ; [17]
        move.l     (r1),d2              ; [19]
        move.l     d2,<_c               ; [19]
        inc        d0                   ; [20]
        adda      #<4,r1                ; [20]
        adda      #<4,r0                ; [20]
        loopend3
...

```

c. Assembly code with simplified redundant accesses

On the contrary, if such a redundant access is detected and simplified, then the second access is not performed. The value stored is preserved in a register and reused for `_c` assignment instead. The assembly code is optimized as follows:

```

...
DW_5
        sub        d0,d0,d0             ; [14] ;CLR
instruction
        adda      #>-36,sp,r0          ; [17]
        adda      #>-40,sp,r1          ; [15]
        doen3     #<10                 ; [0]
        dosetup3  L5                   ; [0]
        move.w    #<1,d1               ; [0]
        loopstart3
L5
        move.l     d0,(r1)              ; [15]
        move.l     d1,(r0)              ; [17]
        move.l     d0,<_c               ; [19]
        inc        d0                   ; [20]
        adda      #<4,r1                ; [20]
        adda      #<4,r0                ; [20]
        loopend3
L4

```

. . .

Access packing (vectorization)

Goal, definition and scope

a. Multiple moves on the StarCore

The StarCore architecture offers multiple move features, i.e. the possibility to perform several moves from/to registers in one single instruction. Different constraints must be verified to properly use those instructions. They mainly concern:

- the type of data ,
- the alignment of the related memory segment,
- the stride between memory slots,
- the relation of domination between successive accesses,
- ...

Those constraints are related with the number of moves to be performed at the same time (2 or 4). Those multiple move instructions can be considered as a specific type of vectorization.

b. Multiple moves from the compiler point of view

Addressing those instructions in a relevant and efficient way is a true compiler concern, especially if the opportunity to use them occurs in a loop. To reach that goal, the compiler must thus verify the set of constraints mentioned above.

The sequences of memory accesses, described in the previous chapter, contain most of the information required to reach that goal. It must be completed by data concerning alignment of memory segments, which is partly retrieved using both symbol tables and information on data-flow. Pragmas may be helpful to ensure alignment of array address provided as function arguments for instance.

We illustrate the effect of access packing using only the assembly view.

Assembly view

a. Initial source code

We reuse the example that illustrated the transformation of sequential accesses:

```
...
short tab[MAX];
...
for(i=0 ; i<MAX ; ) {
    tab[i] = 0 ;
    i++;
    tab[i] = 1;
    i++;
}
...
```

b. Assembly code without packing

After the detection and transformation of the sequence, the code was transformed as follows:

```
...
DW_3
    adda    #>-22,sp,r0          ; [18]
    adda    #>-24,sp,r1          ; [16]
    doen3   #<5                  ; [0]
    dosetup3 L5                  ; [0]
    sub     d0,d0,d0             ; [0] ; CLR
instruction
    move.w   #<1,d1              ; [0]
    loopstart3
L5
    move.w   d0,(r1)              ; [16]
    move.w   d1,(r0)              ; [18]
    adda     #<4,r1               ; [19]
    adda     #<4,r0               ; [19]
    loopend3
L4
...
```

c. Assembly code with packing

The former sequence matches several constraints:

Induction-Related Loop Optimizations

Access packing (vectorization)

- it concerns short integers,
- it concerns an aligned array,
- initial value of "i" variable when first entering the loop is known at compile time, and corresponds with an aligned memory slot (i.e. address is a multiple value of cell size),
- alignment is preserved from one iteration of the loop to the next one,
- it concerns two consecutive memory slots,
- no specific data dependency prevents this packing from being performed,
- ...

When access packing is performed, then the two simple moves are replaced by one double move. Only one address register is necessary instead of two:

```
...
DW_3
    adda      #>-24, sp, r0          ; [16]
    doensh3   #<5                    ; [0]
    sub       d0, d0, d0             ; [0] ; CLR
instruction
    move.w    #<1, d1                ; [0]
    loopstart3
L5
    move.2w   d0:d1, (r0)             ; [16]
    adda      #<4, r0                 ; [19]
    loopend3
...
```

A bestiary

Like sequential accesses, packing can be performed in various cases. We now illustrate such cases.

a. Basic cases

a.1 Packing based on array and multi-step IV ...

```
short array[10];
...
i=0;
for(...) {
    ...
    ... = array[i];
}
```

```
        i++;
        ... = array[i];
        i++;
        ...
    }
    ...
```

a.2 Packing based on array and simple step IV with different static offsets ...

```
short array[10];
...
i=0;
for(...) {
    ...
    ... = array[i];
    ...
    ... = array[i+1];
    ...
    i+=2;
}
...
```

a.3 Packing based on pointer and multi-step IV ...

```
void fct(short ptr[])
{
    #pragma align *ptr 4
    ...
    for(...) {
        ...
        ... = *ptr;
        ptr++;
        ...
        ... = *ptr;
        ptr++;
        ...
    }
    ...
}
...
```

a.4 Packing based on pointer and simple step IV with different static offsets ...

```
void fct(short ptr[])
{
#pragma align *ptr 4
    ...
    for(...) {
        ...
        ... = *ptr;
        ...
        ... = *(ptr+1);
        ptr+=2;
        ...
    }
    ...
}
```

a.5 Packing extracted from complex sequences ...

```
int array[40];
...
for(i=0; i<16;){
    tab[i] = ...;    <----
    i++;             |- first pair of packed
accesses
    tab[i] = ...;    <----
    i+=3;
    tab[i] = ...;
    i+=5;
    tab[i] = ...;
    i+=3;
    tab[i] = ...;    <----
    i++;             |- second pair of packed
accesses
    tab[i] = ...;    <----
    i++;
}
...
```

a.6 Packing with interleaved accesses on different arrays

```
...
int array1[10];
int array2[10];
...
for(i=0; i<5; i+=2){
    acc1=L_add(acc1, L_mult(array1[i],
array2[i]));
    acc2=L_add(acc2, L_mult(array1[i+1],
array2[i+1]));
}
...
```

b. Case of arrays in data structures

b.1 Arrays in structures The first case concerns to arrays located in structures. Let us consider the code below. One defines the structured type (Struct_tab). The data structure contains an array of integer (tab).

```
...
typedef struct Type_struct_tab *Pt_struct_tab;
typedef struct Type_struct_tab {
    long    x;
    short   tab[MAX];
} Struct_tab;
...
```

The successive accesses to the content of the array is now packed:

```
...
Struct_tab array_tab;
...
for(ind1=0 ; ind1<MAX/2; ind1+=2) {
    array_tab.tab[ind1]=ind1;
    array_tab.tab[ind1+1]=ind1+1;
}
...
```

b.2 Simple fields in structures The second case concerns packed accesses to successive fields in structures. We refer to a structure that describes a complex data type:

```
...
typedef struct Type_cplx *Pt_cplx;
typedef struct Type_cplx {
    short    re;
    short    im;
} Struct_cplx;
...
```

The accesses to the fields of the array is now detected as a packed access:

```
...
Struct_cplx array[10];
...
for(ind1=0 ; ind1<MAX/2; ind1+=2) {
    array[ind1].re=0;
    array[ind1].im=1;
}
...
```

c. Cases of multi-dimensional accesses

In some cases, multidimensional accesses can be packed as well, even if it is a more complex issue.

The example below illustrates the access to a linearized 4x4 array. Moreover, the code is structured as a two levels nest of loops. The pointer used for actual access is modified both in inner and outer loops. An accurate analysis shows that the alignment is preserved by the modification of the pointer in outer loop. Thus, packing the four accesses to consecutive columns is legal:

```
...
void fct(..., short* block, ...)
{
    #pragma align *block 8
    ...
    int k, i;
    ...
    p_b = block;
```

```
for(k=0; k<4; k++) {  
    ...  
    for(i=0; i<4; i++) {      /*i is the index of  
the row*/  
        ... = *(p_b+(4*i)+0);    /*1st columns  
accessed in row*/  
        ... = *(p_b+(4*i)+1);    /*2nd columns  
accessed in row*/  
        ... = *(p_b+(4*i)+2);    /*3rd columns  
accessed in row*/  
        ... = *(p_b+(4*i)+3);    /*4th columns  
accessed in row*/  
    }  
    ...  
    p_b+=16; /*this modification to change row*/  
    ...  
}  
...
```

Cross-loop Induction

- [Introduction](#)
- [A bestiary](#)

Introduction

Description of the problem and need

The induction process described in previous chapters only deals with one loop at a time. Such a process mainly optimizes innermost loops. As a consequence it leads to an often sub-optimal assembly code.

The high-level optimizer of the CodeWarrior for StarCore compiler involves powerful cross-loop induction mechanisms that go beyond those limitations. It optimizes loop nests as a whole, instead of single loops.

Let us illustrate this on a first simple example.

Illustration: trivial access to a matrix

a. Initial source code

We consider a full access to a two-dimensional matrix. The corresponding C-code is as follows:

```
...
short matrix[20][10];
...
for(ind1=0; ind1<20; ind1++) {
    for(ind2=0; ind2<10; ind2++) {
        matrix[ind1][ind2]=0;
    }
}
...
```

b. Assembly code obtained with single loop induction process

The code obtained thanks to the induction process described above is as follows:

```
...
DW_2
    move.w    #400,r6                ;[7]
    nop                          ;[0] AGU stall
    adda      r6,sp                  ;[7]
DW_5
    move.w    #<0,r0                ;[16]
    doen2     #<20                  ;[0]
    dosetup2  L10                   ;[0]
    sub       d0,d0,d0              ;[0];CLR
instruction
    loopstart2
L10
    adda      #>-400,sp,r1           ;[16]
    adda      r0,r1                 ;[16]
    doensh3   #<10                  ;[0]
    nop                          ;[0] L_D_3
    loopstart3
L9
    move.w    d0,(r1)               ;[16]
    adda      #<2,r1                ;[15]
    loopend3
L6
```



```

        adda      #<20, r0                ; [14]
        nop                      ; [0]  L_L_4
        loopend2
L8
...

```

One notice that two different address registers are used:

- register \$r0 is used to store the initial address of each row. It is updated in outer loop,
- register \$r1 is used to actually access each row. It is updated in inner loop. Its initial value is computed in outer loop (i.e. each time outer loop is iterated), using register \$r0.

c. Optimized assembly code with cross-loop induction

This trivial case could lead to a more efficient code. Namely, this code contains redundancy: this full matrix access could be performed using one single address register. Moreover because of the linear representation of the arrays in memory, register does not need to be updated in outer loop. The expected code in this case is as follows:

```

...
DW_5
        adda      #>-400, sp, r0          ; [0]
        doen2     #<20                    ; [0]
        dosetup2  L10                     ; [0]
        sub       d0, d0, d0              ; [0] ; CLR
instruction
        loopstart2
L10
        doensh3   #<10                    ; [0]
        nop                      ; [0]  L_D_3
        loopstart3
L9
        move.w    d0, (r0)                ; [16]
        adda      #<2, r0                 ; [15]
        loopend3
L6
        nop                      ; [0]  L_N_1
        nop                      ; [0]  L_L_4

```

```
        loopend2
L8
    ...
```

In other words the actual pointer used to go through a complete matrix evolve linearly, as illustrated in the figure below. One also notices that the number of instructions in inner loop is unchanged. The outer loop now contains any actual instruction: the nest is now a perfect loop nest. Only one address register is needed instead of 2.

Figure 2: two-dimensional matrix access: linear storage and access

d. Result with further loop collapse

In this case, loop nest can even be collapsed. As we shall explain later, the code delivered by CodeWarrior for StarCore is optimized as follows:

```
    ...
    DW_5
        adda    #>-400, sp, r0            ; [0]
        move.w  #200, r1                  ; [0]
        nop                                ; [0] AGU stall
        doensh3 r1                        ; [0]
        sub     d0, d0, d0                ; [0] ; CLR
    instruction
        loopstart3
    L9
        move.w  d0, (r0)                  ; [16]
        adda    #<2, r0                   ; [15]
        loopend3
    L8
    ...
```

Using this loop collapse, only one hardware-loop is needed instead of 2. The code is also much more compact.

Generalization and scope

a. Linear domains scope

The example is a very specific and trivial case. However such an approach can be generalized to any multi-dimensional access performed in loop nests, provided that a set of properties are verified. Amongst those properties the two main ones are the following:

- loop bounds are either invariant or induction variables in enclosing loop. This corresponds to the description of linear iteration spaces,
- loop is not bypassed or the number of bypassed iterations can be assessed at compile time.

Those conditions correspond with many cases of loop nests encountered in DSP applications. The cases that are handled by CodeWarrior are the following ones:

- square and rectangular (i.e. block in a matrix),
- triangular (sometimes truncated),
- diagonal,
- trapezoidal.

The figure below illustrates those cases for a two-dimensional iteration space, i.e. for two consecutive loops of a nest.

Figure 3: access patterns considered

b. High-dimension, mixed nests, perfect loop nests

The number of dimensions (i.e. depth of the nest) is not limited.

The compiler can handle loop nests with different successive domain shapes. For instance, triangular access to successive layers in a cube can be processed and optimized by the compiler. This case is illustrated in the figure below.

Moreover, even if all the examples used involve perfect loop nests, the approach also handles any kind of nest, either perfect or not. The only restrictions concern the way induction variables can be reused across the nest.

...

```

    for(i=0; i<MAX; i++) {
        for(j=0; j<MAX; j++) {
            for(k=0; k<j; k++) {
                ...tab[i][j][k]...;
            }
        }
    }
    ...

```

Figure 4: example of three-dimensions mixed case

A bestiary

Partial (block) matrix access

a. Initial source code

When only a sub-block of the matrix is accessed. The C code can be as follows:

```

...
short matrix[20][10];
...
for(i=0; i<20; i++) {
    for(j=2; j<5; j++) {
        matrix[i][j]=0;
    }
}
...

```

b. Optimized assembly code

In this case the traversal is no longer linear. The pointer must jump from the end of one row to the beginning of next one. The cross-loop optimizer still succeeds in monitoring access across the nest using one single address register. The corresponding assembly code is as follows:

```

...
DW_5
    adda      #>-396,sp,r0          ; [0]
    doen2    #<20                  ; [0]
    dosetup2 L10                   ; [0]

```

```

        sub      d0,d0,d0                ; [0] ;CLR
instruction
    loopstart2
L10
    doensh3     #<3                      ; [0]
    nop                     ; [0]  L_D_3
    loopstart3
L9
    move.w      d0,(r0)                  ; [16]
    adda        #<2,r0                   ; [15]
    loopend3
L6
    adda        #<14,r0                  ; [0]
    nop                     ; [0]  L_L_4
    loopend2
L8
    ...

```

Triangular access

a. Initial source code

Let us now consider a two-dimension triangular access. The domain here is a superior triangular one:

```

...
short matrix[MAX][MAX];
...
for(i=i; i<MAX; i++) {
    for(j=i; j<MAX; j++) {
        matrix[i][j]=0;
    }
}
...

```

Figure 5: shifted triangular domain

b. Un-optimized assembly code

The code obtained thanks to a single-loop induction process is as follows:

```

...
DW_5
    sub        d0,d0,d0                ; [16] ;CLR
instruction
    sub        d0,d0,d1                ; [15] ;CLR
instruction
    sub        d0,d0,d2                ; [16] ;CLR
instruction
    doen2      #<10                    ; [0]
    dosetup2   L10                     ; [0]
    loopstart2
L10
    cmpgt.w    #<9,d1                  ; [15]
    jt         L4                       ; [15]
    move.l     d1,r0                    ; [0]
    move.w     #<10,r1                  ; [0]
    nop                            ; [0] AGU stall
    suba       r0,r1                    ; [0]
    add        d2,d0,d3                ; [16]
    move.l     d3,r2                    ; [16]
    adda       #>-200,sp,r3              ; [16]
    adda       r3,r2                    ; [16]
    doensh3    r1                       ; [0]
    sub        d0,d0,d3                ; [0] ;CLR
instruction
    loopstart3
L9
    move.w     d3,(r2)                  ; [16]
    adda       #<2,r2                    ; [15]
    loopend3
L6
L4
    add        #<2,d2                    ; [14]
    inc        d1                       ; [14]
    add        #<20,d0                  ; [14]
    loopend2
...

```

One notices that:

- inner loop contains 2 instructions, and outer loop contains numerous instructions,
- inner loop bypass test is still present, even though it is never active,
- computing the address across the nest requires both data and address registers. A large amount of data registers is used.

c. Optimized assembly code

Cross-loop induction process delivers the code below:

```
...
DW_5
    sub      d0,d0,d0                ; [15] ; CLR
instruction
    adda     #>-200,sp,r0             ; [0]
    move.w   #<2,r1                  ; [0]
    doen2    #<10                    ; [0]
    dosetup2 L10                     ; [0]
    sub      d0,d0,d1                ; [0] ; CLR
instruction
    loopstart2
L10
    move.l   d0,r2                   ; [0]
    move.w   #<10,r3                 ; [0]
    nop                      ; [0] AGU stall
    suba     r2,r3                   ; [0]
    doensh3   r3                     ; [0]
    nop                      ; [0] L_D_3
    loopstart3
L9
    move.w   d1,(r0)                 ; [16]
    adda     #<2,r0                  ; [15]
    loopend3
L6
    adda     r1,r0                   ; [0]
    inc      d0                      ; [14]
    adda     #<2,r1                  ; [0]
    loopend2
L8
...
```

One notices the following improvements:

- the number of instructions in outer loop is much smaller,
- the accurate optimizer analysis detected and removed useless inner-loop bypass test,
- the address used for triangular access is monitored across the nest using \$r0, which is set before loop nest and inducted in both inner and outer loop. Its induction in outer loop is a second order one, i.e. its step is an induction variable in outer loop, stored in \$r1. As a consequence, the need for data register is reduced.

Shifted triangular access

a. Initial source code

Let us now consider a two-dimension triangular access. The domain here is an inferior triangular one. This inner loop highest bound is now different from the main diagonal line (the bound is "shifted" to the right):

```
...
short matrix[MAX] [MAX];
...
for(i=3; i<MAX; i++) {
    for(j=0; j<=i-3; j++) {
        matrix[i] [j]=0;
    }
}
...
```

Figure 6: shifted triangular domain

b. Un-optimized assembly code

The code obtained thanks to a single-loop induction process is as follows:

```
...
DW_5
    move.w    #<60, r0                ; [16]
    sub      d0, d0, d0              ; [15] ; CLR
instruction
```

```

        doen2      #<7                                ; [0]
        dosetup2   L10                                ; [0]
        loopstart2
L10:    tstge      d0                                ; [15]
        jf         L4                                ; [15]
        addnc.w    #<1,d0,d1                          ; [0]
        adda       #>-200,sp,r1                       ; [16]
        adda       r0,r1                             ; [16]
        doensh3    d1                                ; [0]
        sub        d0,d0,d1                          ; [0] ;CLR
        instruction
        nop                               ; [0]  L_D_3
        loopstart3
L9:     move.w     d1,(r1)                          ; [16]
        adda       #<2,r1                             ; [15]
        loopend3
L6:
L4:     inc        d0                                ; [14]
        adda       #<20,r0                            ; [14]
        nop                               ; [0]  L_C_1
        loopend2
L8:
...

```

One remarks the same features than in the former example, especially concerning bypass test.

c. Optimized assembly code

Cross-loop induction process delivers the code below:

```

...
DW_5:  sub        d0,d0,d0                          ; [15] ;CLR
        instruction
        adda       #>-140,sp,r0                      ; [0]
        move.w     #<18,r1                           ; [0]
        doen2      #<7                                ; [0]
        dosetup2   L10                                ; [0]

```

Induction-Related Loop Optimizations

A bestiary

```
        sub      d0,d0,d2                ; [0] ;CLR
instruction
    loopstart2
L10
    addnc.w     #<1,d0,d1                ; [0]
    doensh3     d1                        ; [0]
    nop         ; [0] L_D_3
    nop         ; [0] L_D_3
    loopstart3
L9
    move.w      d2,(r0)                   ; [16]
    adda        #<2,r0                    ; [15]
    loopend3
L6
    adda        r1,r0                     ; [0]
    inc         d0                        ; [14]
    suba        #<2,r1                    ; [0]
    loopend2
L8
    ...
```

The same kind of improvements can be remarked, even if the gain in terms of instructions is lower in this example.

Truncated triangular access

a. Initial source code

Triangular access may be truncated. Let us now consider such a truncated two-dimension triangular access:

```
...
short matrix[MAX][MAX];
...
for(i=3; i<MAX; i++) {
    for(j=0; j<i; j++) {
        matrix[i][j]=0;
    }
}
...
```

Figure 7: truncated triangular domain

b. Un-optimized assembly code

The code obtained thanks to a single-loop induction process is as follows:

```
...
DW_5
    move.w    #<3,d0                ; [14]
    move.w    #<60,r0              ; [16]
    doen2     #<7                  ; [0]
    dosetup2  L10                  ; [0]
    loopstart2
L10
    tstgt     d0                   ; [15]
    jf        L4                  ; [15]
    move.l    d0,r1                ; [0]
    adda      #>-200,sp,r2          ; [16]
    adda      r0,r2                ; [16]
    doensh3   r1                   ; [0]
    sub       d0,d0,d1             ; [0] ; CLR
instruction
    loopstart3
L9
    move.w    d1,(r2)              ; [16]
    adda      #<2,r2               ; [15]
    loopend3
L6
L4
    inc       d0                   ; [14]
    adda      #<20,r0              ; [14]
    nop                          ; [0]  L_C_1
    loopend2
L8
...
```

One remarks the same features than in the former example, especially concerning bypass test.

c. Optimized assembly code

Cross-loop induction process delivers the code below:

```
...
DW_5
    move.w    #<3,d0                ; [14]
```

Induction-Related Loop Optimizations

A bestiary

```
        adda      #>-140,sp,r0          ; [0]
        move.w    #<14,r1              ; [0]
        doen2     #<7                  ; [0]
        dosetup2  L10                  ; [0]
        sub       d0,d0,d1             ; [0] ;CLR
instruction
    loopstart2
L10
    move.l       d0,r2                  ; [0]
    nop          ; [0] AGU stall
    doensh3      r2                    ; [0]
    nop          ; [0] L_D_3
    loopstart3
L9
    move.w       d1,(r0)                ; [16]
    adda         #<2,r0                 ; [15]
    loopend3
L6
    adda         r1,r0                  ; [0]
    inc          d0                    ; [14]
    suba         #<2,r1                 ; [0]
    loopend2
L8
...
```

The same kind of improvements can be remarked.

Mixed high-dimension case

a. Initial source code

We now illustrate how the optimizer handles cases with both higher dimensions and mixed domain shapes. The example corresponds with a tree-dimension mixed square/triangular iteration space:

```
...
for(i=0; i<MAX; i++) {
    for(j=0; j<MAX; j++) {
        for(k=0; k<=j; k++) {
            ...tab[i][j][k]...;
        }
    }
}
```

...

Figure 8: tree-dimension mixed case

b. Un-optimized assembly code

The code obtained thanks to a single-loop induction process is as follows:

```

...
DW_2
    move.w    #2000,r6                ;[7]
    nop                      ;[0] AGU stall
    adda      r6,sp                ;[7]
DW_5
    sub       d0,d0,d0              ;[17];CLR
instruction
    doen1     #<10                  ;[0]
    dosetup1  L15                   ;[0]
    loopstart1
L15
    sub       d0,d0,d1              ;[15];CLR
instruction
    move.l    d0,r0                 ;[17]
    doen2     #<10                  ;[0]
    dosetup2  L14                   ;[0]
    falign
    loopstart2
L14
    tstge     d1                    ;[16]
    jf        L6                    ;[16]
    addnc.w   #<1,d1,d2             ;[0]
    adda      #>-2000,sp,r1          ;[17]
    adda      r0,r1                 ;[17]
    doensh3   d2                    ;[0]
    sub       d0,d0,d2              ;[0];CLR
instruction
    nop                      ;[0] L_D_3
    loopstart3
L13
    move.w    d2,(r1)               ;[17]
    adda      #<2,r1                ;[16]
    loopend3
L8
L6

```

Induction-Related Loop Optimizations

A bestiary

```
        inc      d1                ; [15]
        adda     #<20, r0          ; [15]
        nop                        ; [0]  L_C_1
        loopend2
L10     addnc.w  #200, d0, d0       ; [14]
        loopend1
L12
...

```

c. Optimized assembly code

Cross-loop induction process delivers the code below:

```
...
DW_5
    sub      d0, d0, d0            ; [15] ; CLR
instruction
    move.w   #<18, d1              ; [0]
    adda     #>-2000, sp, r0       ; [0]
    doen1    #<10                  ; [0]
    dosetup1 L15                   ; [0]
    sub      d0, d0, d3            ; [0] ; CLR
instruction
    loopstart1
L15
    doen2    #<10                  ; [0]
    dosetup2 L14                   ; [0]
    falign
    loopstart2
L14
    addnc.w  #<1, d0, d2           ; [0]
    doensh3  d2                    ; [0]
    nop      ; [0]  L_D_3
    nop      ; [0]  L_D_3
    loopstart3
L13
    move.w   d3, (r0)              ; [17]
    adda     #<2, r0               ; [16]
    loopend3
L8
    move.l   d1, r1                ; [0]
    nop      ; [0]  AGU stall

```

```
        adda      r1,r0                ; [0]
        inc       d0                  ; [15]
        sub       #<2,d1              ; [0]
        loopend2
L10:
        tfr       d3,d0                ; [0]
        move.w    #<18,d1             ; [0]
        loopend1
L12:
        . . .
```

The same kind of improvements can be remarked. When MAX equals 10, then the gain in terms of number of cycles with optimization level 1 is about 8,8% (2147 instead of 2354). With optimization level 3, the gain is 7,5% (1364 cycles against 1474).

Induction-Related Loop Optimizations

A bestiary

Loop Restructuring and Reordering

- [Definitions and Scope](#)
- [Loop-Collapse](#)
- [Loop Peeling](#)
- [Loop Unrolling](#)
- [Partial Summation](#)

Definitions and Scope

- [Some definitions](#)
- [Features of CodeWarrior for StarCore](#)

Some definitions

Contrary to the optimizing techniques presented in the previous chapters, loop restructuring and reordering transformations modify the structure of the loop and/or the way computations are performed inside loop. Many different techniques are known. A (non-exhaustive) list can be found in [BGS94] and [KST93] for instance.

Loop restructuring (like unrolling and pipelining) preserves the order of computation, whereas loop reordering (like interchange and jamming) changes the relative order of execution of the iterations of a loop nest. As a consequence those techniques require:

- for reordering techniques: a proof of their legality, from the data dependency point of view,
- for both restructuring and reordering: an accurate control so as to apply them in relevant cases, i.e. only when a gain can be expected. For some of those transformation this is a hard point: transformation may strongly interact with other optimization

techniques, control may require information which is not easily available at the time the transformation is performed (register pressure, intrinsic code parallelism, ...)...

Features of CodeWarrior for StarCore

In former versions of CW for StarCore, only loop collapse and loop peeling were performed in an automatic way. Unrolling and unroll-and-jam [Din96] were also available. However, the default configuration did not activate them. They could only be enabled by means of pragmas.

New release now includes an automatic control of unrolling, which is thus applied only if it improves code speed. Three restructuring techniques are thus available and controlled in an automatic way:

- loop collapse, which is used to reduce perfect loop nests to a single loop whenever it is possible,
- loop peeling, which is used to transform wrap-around variables,
- loop unrolling.

Loop unroll-and-jam remains available thanks to pragma.

Loop-Collapse

- [Overview and goal](#)
- [Assembly view and result](#)

Overview and goal

In order to improve the result of full-matrix access regularization, we implemented a simple collapse of perfect loop nests. This function let the loop structure unchanged, but it modifies the actual loop count of the nest. Loop structure is modified during code-generation stage. Pure linear accesses are thus realized thanks to one single loop.

This simple transformation makes no use of IV information. It is performed when all the loops of an entry have been transformed by induction process. It consists in:

- removing all useless induction instructions, so as to expose perfect loop nests,

- detecting and collapsing perfect loop nests in a recursive way (innermost loops are processed first).

In other words if two consecutive loops of a nest are found to form a perfect nest, then the outer loop count is multiplied by inner-loop one, and inner-loop-count is set to one. This process is performed recursively from innermost loops towards outer-most ones. During code-generation, useless back-edges of the flow graphs are removed.

Let us consider the code below:

```
...
short matrix[MAX][MAX];
...
for(ind1=0; ind1<MAX; ind1++) {
    for(ind2=0; ind2<MAX; ind2++) {
        matrix[ind1][ind2]=0;
    }
}
...
```

This code is first transformed by loop normalization and cross-loop induction process. One obtains:

```
...
short matrix[MAX][MAX];
...
pt=matrix;
...
ind1=0;
for(lc1=MAX; lc1>0; lc1--) {
    ind2=0;
    for(lc2=MAX; lc2>0; lc2--) {
        *pt=0;
        pt = pt + 2;
        ind2++;
    }
    ind1++;
}
...
```

Useless induction variables are simplified:

Loop Restructuring and Reordering

Assembly view and result

```
...
short matrix[MAX][MAX];
...
pt=matrix;
...
for(lc1=MAX; lc1>0; lc1--) {
    for(lc2=MAX; lc2>0; lc2--) {
        *pt=0;
        pt = pt + 2;
    }
}
...
```

As loop nest is now a perfect one, it can be collapsed as follows:

```
...
short matrix[MAX][MAX];
...
pt=matrix;
...
for(lc1=MAX*MAX; lc1>0; lc1--) {
    *pt=0;
    pt = pt + 2;
}
...
```

Assembly view and result

Optimized assembly code with cross-loop induction

The example of the full matrix access is reused here. The code transformed by cross-loop induction process is presented below:

```
...
DW_5
    adda      #>-200,sp,r0          ; [0]
    doen2    #<10                  ; [0]
    dosetup2  L10                  ; [0]
    sub      d0,d0,d0              ; [0] ; CLR
instruction
    loopstart2
L10
```

```
doensh3    #<10                ; [0]
nop        ; [0]  L_D_3
loopstart3
L9
move.w     d0, (r0)             ; [17]
adda       #<2, r0              ; [16]
loopend3
L6
nop        ; [0]  L_N_1
nop        ; [0]  L_L_4
loopend2
L8
...
```

Result with loop collapse

In this case, loop nest can be collapsed. The code delivered by the compiler is optimized as follows:

```
...
DW_5
adda       #>-200, sp, r0        ; [0]
move.w     #100, r1              ; [0]
nop        ; [0]  AGU stall
doensh3    r1                    ; [0]
sub        d0, d0, d0            ; [0] ; CLR
instruction
loopstart3
L9
move.w     d0, (r0)             ; [17]
adda       #<2, r0              ; [16]
loopend3
L8
...
```

Using this loop collapse, the number of cycles for this loop lower, and only one hardware-loop is needed instead of 2.

Loop Peeling

- [Overview and goal](#)

- [Assembly view and result](#)

Overview and goal

Loop peeling is needed to process wrap-around variables. Although a generic peeling is actually implemented, peeling loop once is sufficient in this context.

The transformation simply consists in:

- checking that the loop can be peeled if restrictions are brought to the process,
- replicating loop body before the loop itself as many times as needed, depending on the peeling factor,
- modify the control of the loop by subtracting the peeling factor to initial value of loop count.

It is controlled by the detection of at least one wrap around variables in the loop. It is performed after detection of IV in loop, and before any transformation of them.

A dummy example of code to be peeled because of a wrap around variable is provided below:

```
...
wrap = f (...);
...
i=0;
for(...) {
    ...array[wrap]...;
    wrap = i;
    i++;
}
...
```

When loop is peeled once, then one obtain the following code:

```
...
wrap = f (...);
...
i=0;
/** first iteration peeled */
...array[wrap]...;
wrap = i;
i++;
```

```
/******  
for(...) {  
    ...array[wrap]...;  
    wrap = i;  
    i++;  
}  
...
```

Assembly view and result

Loop peeling is performed to allow the induction process to be applied to wrap-around variables. In other words the assembly code of the related loop is then similar to that of a loop after the simplification of regular induction variables.

Loop Unrolling

- [Overview and goal](#)
- [Assembly view and result](#)

Overview and goal

Advantage of unrolling

Unrolling is a well-known transformation. It simply consists in replicating the body of a loop some number of times. The main advantages of high-level unrolling for a machine like the StarCore results from the possibility to expose some patterns that can be transformed by the high-level loop optimizer:

- a potential increase of the number of opportunities to make use of the multiple (double or quadruple) moves (load/store) instructions,
- a potential reduction of the number of memory accesses thanks to new redundant memory accesses,

Of course it may also induce an improvement of the intrinsic code parallelism. Thus the scheduler can then do a better job.

Need for automatic control

However, expecting a certain gain thanks to high-level unrolling performed without any further control is highly speculative. As we told above, all the gains are "potential" ones.

As some of the expected gains result from more numerous opportunities for high-level optimizations (multiple moves, redundant memory accesses, ...), then it is clear that unrolling should be performed before loop is optimized. Thus one can hardly predict the actual improvement of performances at this moment. Nothing is known about actual code (register allocated, instructions selected, opportunities for parallelization, ...). In some cases for instance, unrolling may lead to an increased register pressure. In worst cases, one may reach the limit beyond which memory spills must be introduced inside loop.

So, a smart control is definitely needed to avoid this trap. This control must be aware of both the high-level representation of the code and low level features of the target architecture (possible mappings, scheduling capabilities, ...)

In the current version of CW for StarCore, loop unrolling is controlled by a high level scheduler, that assesses the best unrolling factor for a given loop. To make a relevant decision, it estimates the effect of code generation and parallelization on the intermediate code of this loop.

Scope and restrictions in current release

Automatic unrolling is currently restricted to the following cases:

- loop must have a static loop count (i.e. known at compile time),
- loop count must be a multiple value of unrolling factor,
- loop body must not contain any control flow (i.e. one block in loop body),
- loop body must not contain any call to function returning structure,
- loop body must not contain any modulo IV.

The future versions of the compiler will overcome those restrictions.

Enabling automatic unrolling, options

Loop unrolling is enabled thanks to dedicated options of the compiler.

This option specifies the maximal value of unrolling factor. This value is expected to be either:

- 0: no unrolling, option is `-u0`, or
- 2: loops either unrolled by 2 or not unrolled, depending on the HLS decision, option is `-u2`, or
- 4: loops unrolled by either 2 or 4 or not unrolled, according to HLS decision, option is `-u4`.

Loops unrolled by means of a local pragma are always unrolled by the specified factor, whatever the HLS decision. Unrolling is always disabled when code is optimized for size (`-Os`).

Assembly view and result

Initial source code

One could present many examples that could illustrate the effects of unrolling. The inner most loop below exposes part of the possible improvements induced by this transformation:

```
...
Word16 y[L_WINDOW];
...
do {
    ...
    if (...) {
        ...
        for (i = 0; i < L_WINDOW; i++) {
            y[i] = shr (y[i], 2);
        }
    }
} while (...);
...
```

Assembly code without unrolling

When unrolling is disabled, the resulting code for the innermost loop can be read between `loopstart3/loopend3` keywords. Some

Loop Restructuring and Reordering

Assembly view and result

instructions related with this loop can be found before and after it due to pipelining (see instruction with [153] as line number). Loop body contains two packets:

```
PL000
[
    mac          d1,d1,d6                ; [146] 1%=1
    adda         #>-504,sp,r0             ; [153] B6
]
[
    cmpeq        d2,d6                   ; [0]
    tfra         r0,r1                   ; [0] B6
]
[
    move.f       (r1)+,d1                 ; [153] 0%=0 B6
    bf           <L21                     ; [148]
]
[
    asrr         #<2,d1                   ; [153] 1%=0
    suba         #<1,r3                   ; [0]
    move.f       #<4,d5                   ; [149]
]
[
    add          d4,d5,d4                 ; [149]
    doensh3      r3                       ; [0]
]
[
    move.w       #<1,d5                   ; [150]
    adda         #<1,r3                   ; [0]
]
    skipls      PL002                     ; [0]
    nop          ; [0] L_C_2
    loopstart3
[
    moves.f      d1,(r0)+                 ; [154] 2%=1
    move.f       (r1)+,d1                 ; [153] 0%=0
]
    asrr        #<2,d1                     ; [153] 1%=0
    loopend3
PL002
    moves.f      d1,(r0)+                 ; [154] 2%=1
L35
```

Assembly code with automatic unrolling:

When automatic unrolling is enabled and maximal unrolling factor set to 4 (-u4 compiler option), then the control strategy determines that loop can be profitably unrolled by 4. The resulting code can be read below. One notices that loop body only contains two packets, including one with two 4-moves. Moreover the pipeliner could perform a much more aggressive transformation:

```
PL000
[
    mac        d4,d4,d3                ; [146] 1%=1
    adda       #>-504,sp,r0            ; [153] B6
]
[
    cmpeq      d5,d3                  ; [0]
    tfra       r0,r2                  ; [0] B6
]
[
    ift add     d1,d7,d1                ; [149] B6
    ifa bf      <L21                    ; [148]
]
[
    move.4f     (r2)+,d8:d9:d10:d11    ; [153] 0%=0
    doensh3     #59                     ; [0] @II2
]
[
    asrr        #<2,d8                  ; [153] 1%=0
    asrr        #<2,d9                  ; [153] 1%=0
    asrr        #<2,d10                 ; [153] 1%=0
    move.w      #<1,d2                  ; [150]
]
    asrr        #<2,d11                  ; [153] 1%=0
    loopstart3
[
    moves.4f    d8:d9:d10:d11,(r0)+    ; [154] 2%=1
    move.4f     (r2)+,d8:d9:d10:d11    ; [153] 0%=0
]
[
    asrr        #<2,d8                  ; [153] 1%=0
    asrr        #<2,d9                  ; [153] 1%=0
    asrr        #<2,d10                 ; [153] 1%=0
    asrr        #<2,d11                 ; [153] 1%=0
]
```

```
loopend3
moves.4f d8:d9:d10:d11, (r0)+ ; [154] 2%=1
L35
```

Partial Summation

- [Overview and goal](#)
- [Assembly view and result](#)

Overview and goal

Principle of the transformation

In some cases data dependency may prevent the parallelizer from doing an optimal job. This may especially occur when a multiple accumulation is performed, like in the code below:

```
...
accu=0;
for(...) {
    accu=accu+tab[i++];
    ...
    accu=accu+tab[i++];
    ...
}
...
```

A very simple transformation may suppress the data dependency carried by the accu variable, without modifying the semantics of the code. It consists in splitting the sequence of accumulations as follows:

```
...
accu1=0;
accu2=0;
for(...) {
    accu1=accu1+tab[i++];
    ...
    accu2=accu2+tab[i++];
    ...
}
accu=accu1+accu2;
```

...

This transformation, which is also called partial summation, is well known by digital signal programmers, who often perform it manually either in the C code or in the assembly code.

Restriction and scope

a. Ensuring the safety of the transformation: saturation and monotonicity

In the context of the CW compiler for StarCore, this transformation can be performed automatically by the high-level optimizer. This is especially likely to increase the benefic effect of unrolling.

However it must be applied with some precautions, so that the semantics of the code is preserved:

- if the accumulator is an integer (either signed or unsigned), then a sufficient condition consists in combining accumulations involving either additions or subtraction or both, like that below:

```
acc=acc+/-var
```

- if the accumulator is a fractional, then a sufficient condition consists in only processing monotonic accumulation instructions that can not reach saturation at any stage of the calculation. This can be done by only transforming quadratic accumulation involving either addition or subtraction :

```
acc=Add/Sub(acc, Mult(var, var))
```

b. Ensuring the safety of the transformation: other accesses to accumulator inside loop

Moreover the transformation is performed iff the variable used as accumulator is neither redefined nor reused outside the accumulation instructions inside the loop. This holds for both direct and indirect (through aliases) accesses.

c. Ensuring that transformation is relevant, control strategy

To perform this transformation in relevant cases only, then it is disabled in the following cases:

- if the accumulation instruction is detected as an induction one,
- if the accumulator is neither an integer nor a fractional data type accessed in a direct way (pointers are excluded for instance),

- as the transformation results in a higher number of registers to be used inside loop, then it is performed iff the assessed register pressure is low enough.

Assembly view and result

a. First example: case of an immediate accumulator

a.1 Initial source code This example is based on a typical loop that can be found in many applications. It aims to compute an energy value. This section presents how the partial summation process may improve code speed when combined with CW automatic unrolling.

```
...
Word16 y[L_WINDOW];
Word32 sum;
...

do {
    ...
    sum = 0L;
    for (i = 0; i < L_WINDOW; i++) {
        sum = L_mac (sum, y[i], y[i]);
    }
    ...
} while(...)
...
```

a.2 Optimized assembly code with unrolling, without any partial summation In order to measure the actual benefit of partial summation within the context of an automatically unrolled loop, one must first notice that the decision of the scheduler concerning the best possible unrolling factor will take the potential effect of partial summation into account. To illustrate this, one provide the code delivered by the compiler when option -u4 and -O3 are set (maximal unrolling factor is 4, code is parallelized). In this case the control strategy of unrolling (i.e. High-Level Scheduler) determines that the loop must not be unrolled at all. The core of the loop is visible between flags loopstart3 and loopend3. The effect of code parallelization and pipelining is reduced (instruction related with the loop are flagged with [145] line number):

...

```

L17
[
    sub        d0,d0,d2                ;[141];CLR instruction
    sub        d0,d0,d3                ;[142];CLR instruction
    suba       #<1,r1                  ;[0]
    adda       #>-504,sp,r0             ;[145]
]
doensh3   r1                          ;[0]
[
    adda       #<1,r1                  ;[0]
    move.f     (r0)+,d4                ;[145]  0%=0
]
    skipls    PL000                   ;[0]
    loopstart3
[
    mac        d4,d4,d3                ;[146]  1%=1
    move.f     (r0)+,d4                ;[145]  0%=0
]
    loopend3
PL000
[
    mac        d4,d4,d3                ;[146]  1%=1
    adda       #>-504,sp,r0             ;[153]  B6
]
...

```

If the control strategy of unrolling is disabled and unrolling by 4 forced, the code delivered is as follows. Because of data dependency, the core of the loop contains 4 packets:

```

...
L17
[
    sub        d0,d0,d3                ;[142];CLR instruction
    sub        d0,d0,d2                ;[141];CLR instruction
    adda       #>-504,sp,r0             ;[145]
    doen3      #59                     ;[0]  @II4
]
[
    move.4f    (r0)+,d4:d5:d6:d7        ;[145]  0%=0
    dosetup3   L54                      ;[0]
]
    mac        d4,d4,d3                ;[145]  1%=0

```

Loop Restructuring and Reordering

Assembly view and result

```
    mac      d5,d5,d3          ; [145] 2%=0
    mac      d6,d6,d3          ; [145] 3%=0
    falign
    loopstart3
L54
[
    mac      d7,d7,d3          ; [146] 4%=1
    move.4f  (r0)+,d4:d5:d6:d7 ; [145] 0%=0
]
    mac      d4,d4,d3          ; [145] 1%=0
    mac      d5,d5,d3          ; [145] 2%=0
    mac      d6,d6,d3          ; [145] 3%=0
    loopend3
[
    mac      d7,d7,d3          ; [146] 4%=1
    move.l   #2147483647,d4    ; [148]
]
...
```

a.3 Optimized assembly code with both automatic unrolling and partial summation The code below illustrates the combined effect of automatic unrolling and partial summation (optimization level is -O3). One notice that pipeliner can make a better job (many instructions flagged [145] before/after loop) and also that the core of the loop now contains only one packet:

```
...
L17
[
    sub      d0,d0,d4          ; [145];CLR instruction
    sub      d0,d0,d8          ; [141];CLR instruction
    adda     #>-480,sp,r5       ; [145]
]
[
    sub      d0,d0,d5          ; [145];CLR instruction
    sub      d0,d0,d7          ; [145];CLR instruction
    sub      d1,d1,d6          ; [145];CLR instruction
    doensh3  #59                ; [0] @II1
    move.4f  (r5)+,d0:d1:d2:d3 ; [145] 0%=0
]
    nop                        ; [0] L_D_3
    loopstart3
```

```
[
    mac          d0,d0,d5                ; [145] 1%=1
    mac          d1,d1,d7                ; [145] 1%=1
    mac          d2,d2,d6                ; [145] 1%=1
    mac          d3,d3,d4                ; [146] 1%=1
    move.4f      (r5)+,d0:d1:d2:d3      ; [145] 0%=0
]
    loopend3
[
    mac          d0,d0,d5                ; [145] 1%=1
    mac          d1,d1,d7                ; [145] 1%=1
    mac          d2,d2,d6                ; [145] 1%=1
    mac          d3,d3,d4                ; [146] 1%=1
    adda         #>-480,sp,r1            ; [153] B6
]
[
    add          d14,d5,d1                ; [145]
    tfra         r1,r6                    ; [0] B6
]
    add          d1,d7,d5                ; [145]
    add          d5,d6,d0                ; [145]
[
    add          d0,d4,d4                ; [145]
    move.4f      (r6)+,d0:d1:d2:d3      ; [153] 0%=0 B6
]
...
```

b. Second example: accumulation in array cells

b.1 Initial source code In many cases accumulation is stored in/read from an array:

```
...
Word16 tabf[SIZE][L_WINDOW];
Word32 accf[SIZE];
...

for(i=0; i<SIZE; i++) {
    accf[i]=0;
    for(j=0; j<L_WINDOW; j++) {
```

Loop Restructuring and Reordering

Assembly view and result

```
        accf[i]=L_mac (accf[i], tabf[i][j],
        tabf[i][j]);
    }
}
...
```

b.2 Optimized assembly code with automatic unrolling, scalarization and partial summation In such cases, an efficient code can be delivered thanks to the successive optimization stages.

b.2.1 Automatic loop unrolling Loop is first unrolled, according to the corresponding option (-u2/-u4). The resulting assembly code is equivalent to the code obtained from the C-code below (-u4 assumed here):

```
...
Word16 tabf[SIZE][L_WINDOW];
Word32 accf[SIZE];
...

for(i=0; i<SIZE; i++) {
    accf[i]=0;
    for(j=0; j<L_WINDOW; ) {
        accf[i]=L_mac (accf[i], tabf[i][j], tabf[i][j]); j++;
        accf[i]=L_mac (accf[i], tabf[i][j], tabf[i][j]); j++;
        accf[i]=L_mac (accf[i], tabf[i][j], tabf[i][j]); j++;
        accf[i]=L_mac (accf[i], tabf[i][j], tabf[i][j]); j++;
    }
}
...
```

b.2.2 Access scalarization As *i* is invariant inside inner-most loop, accesses to array *acc[i]* can then be scalarized. This exposes a sequence of accumulation like in the code below. The corresponding accumulator is variable *t1*:

```
...
Word16 tabf[SIZE][L_WINDOW];
Word32 accf[SIZE];
...

for(i=0; i<SIZE; i++) {
```

```

    accf[i]=0;
    t1=accf[i];
    for(j=0; j<L_WINDOW; j++) {
        t1=L_mac (t1, tabf[i][j], tabf[i][j]);
        t1=L_mac (t1, tabf[i][j], tabf[i][j]);
        t1=L_mac (t1, tabf[i][j], tabf[i][j]);
        t1=L_mac (t1, tabf[i][j], tabf[i][j]);
    }
    accf[i]=t1;
}
...

```

b.2.3 Partial summation, HLS decision, and resulting assembly code Finally this sequence of accumulation can be transformed by means of the partial summation. Moreover the High-Level Scheduler makes the decision to preserve the unrolled version of the loop. Thus the initial C-code listed in section b.1 results in the optimized assembly code below:

```

...
DW_18
[
    doen2      #<16                      ; [0]
    dosetup2   L30                      ; [0]
]
...
DW_20
    move.l     #_accf,r0                  ; [36]
    move.l     #_tabf,r1                  ; [0]
    falign
    loopstart2
L30
[
    sub        d0,d0,d1                   ; [38];CLR instruction
    sub        d0,d0,d2                   ; [38];CLR instruction
    sub        d1,d1,d3                   ; [38];CLR instruction
    move.w     #<0,d0                     ; [36]
    doensh3    #3                         ; [0] @II1
]
[
    move.l     d0,(r0)                    ; [36]
    move.4f    (r1)+,d4:d5:d6:d7          ; [38] 0%=0

```

Loop Restructuring and Reordering

Assembly view and result

```
]
    loopstart3
[
    mac          d4,d4,d0          ; [38] 1%=1
    mac          d5,d5,d1          ; [38] 1%=1
    mac          d6,d6,d2          ; [38] 1%=1
    mac          d7,d7,d3          ; [39] 1%=1
    move.4f      (r1)+,d4:d5:d6:d7 ; [38] 0%=0
]
    loopend3
[
    mac          d5,d5,d1          ; [38] 1%=1
    mac          d4,d4,d0          ; [38] 1%=1
    mac          d6,d6,d2          ; [38] 1%=1
    mac          d7,d7,d3          ; [39] 1%=1
]
    add          d0,d1,d4          ; [38]
    add          d4,d2,d5          ; [38]
    add          d5,d3,d6          ; [38]
    move.l       d6,(r0)+          ; [0]
    loopend2
...

```

Loop Restrictions

- [Limitations Concerning Single-Loop Induction](#)
- [Limitations of Cross-Loop Mechanisms](#)
- [Limitations of Sequential Accesses and Packing](#)
- [Case Study: G729 cor_h Function](#)

Limitations Concerning Single-Loop Induction

- [IV redefinition](#)
- [Ambiguous definition due to function call](#)
- [Multiple conditional induction](#)
- [Second order induction](#)

IV redefinition

Description of the problem

The detection of IV do not cope with redefined variables, as soon as redefinition breaks the inducted behavior:

- non-basic IV with multiple definitions,
- basic IV and non-basic IV with killing redefinition.

The corresponding pieces of code are dumped below.

a. Constant redefinition

```
...
for (...) {
    ...
    ind3 = 2*ind1;
    ...
    ind3 = 0;
    ...
    ind1++;
}
```

...

b. Multiple dependence on basic IV

```
...
for(...) {
    ...
    ind3 = 2*ind1;
    ...
    ind3 = 3*ind1;
    ...
    ind1++;
}
...
```

c. Definition as both basic and non-basic IV

```
...
for(...) {
    ...
    ind3 = 2*ind1;
    ...
    ind3++;
    ...
    ind1++;
}
...
```

5.1.1.2 Possible solution

One should de-correlate the two conflicting definitions by using two different variables. For instance:

```
...
for(...) {
    ...
    ind3 = 2*ind1;
    ...
    ind3_bis = 3*ind1;
    ...
    ind1++;
}
```

...

Ambiguous definition due to function call

Description of the problem

A special case of redefinition can arise when a variable is likely to be modified by a function call:

```
...
for(...) {
    ...
    ... fct(..., &not_basic_iv, ...)...;
    ...
    not_basic_iv=not_basic_iv+1;
    not_non_basic_iv = not_basic_iv * 3;
    ...
}
...
```

In this case the variable cannot be considered as a basic IV. Variables defined thanks to a linear function of this variable can no longer be considered as a non-basic IV.

Important remark:

This limitation is a general one, and also concerns invariant code motion (such a variable whose address is provided as input argument to a function cannot be considered as a loop invariant variable), sequences of memory accesses and packing (such an ambiguous definition may break a sequence), cross-loop mechanism, ...

Possible solution

In this case it may be more efficient to make use of a copy to be passed as function argument. This is possible iff the function does not actually modifies the value of the variable:

```
...
for(...) {
    ...
    aux_var = basic_iv;
```

Loop Restrictions

Multiple conditional induction

```
... fct(..., &aux_var, ...) ...;
...
basic_iv= basic_iv+1;
non_basic_iv = basic_iv * 3;
...
}
...
```

Multiple conditional induction

Description of the problem

Another limitation concerns variables which are inducted several times in different conditional branches of the loop body:

```
...
for(...) {
    ...
    if(...) {
        var = var + 1;
    } else {
        var = var + 2;
    }
    ...
}
...
```

In this case the variable cannot be considered as a basic IV. Variables defined thanks to a linear function of this variable can no longer be considered as a non-basic IV. The same kind of limitation concerns multi-step IV with conditional induction:

```
...
for(...) {
    ...
    if(...) {
        var = var + 1;
    }
    ...
    var = var + 2;
    ...
}
```


...

Possible solution

Today there is no actual way to overcome this limitation.

Second order induction

Description of the problem

Even if the cross-loop mechanism handles second order IV (i.e. induction variable whose step is inducted too), the compiler does not cope with user-defined second order induction. For instance in the example below, "var2" is an induction variable whereas "var1" is not:

```
...
for(...) {
    ...
    var1 = var1 + var2;
    ...
    var2++;
    ...
}
...
```

Possible solution

Today there is no actual way to overcome this limitation.

Limitations of Cross-Loop Mechanisms

- [General restriction on loop steps](#)
- [Reused variables](#)
- [Implicit cross-loop combination](#)
- [Conditional inner loop](#)
- [Bypassed inner loop](#)

General restriction on loop steps

Description of the problem

Cross-loop mechanism only deals with hardware loops. It handles loops which are controlled by variables whose step is:

- either an integer equal to a power of two (including one). In this case both short and long integers are taken into account,
- or a short integer that can be computed using fractional multiplication. In the current version only steps equal to 3, 5 and 7 are accepted.

If step does not fit this constraint, then cross-loop mechanism does not process the loops.

Moreover, if the step fits the constraint, but is not a unit one, then the gain obtained thanks to cross-loop mechanism is lower. This is especially true when a loop contains induction variables whose step is lower than that of the variable used for loop control. Namely, the cross-loop mechanism introduces new instructions to compute the loop count. The formula is, for large inequalities:

$$LC = (High_Bound - Low_Bound + step) / step$$

Or, for strict inequalities:

$$LC = (High_Bound - Low_Bound - 1 + step) / step$$

It then requires some intermediate divisions (or shift operations) which can not be simplified by induction process.

Possible solution

There is no way to overcome this limitation, except if the loop can be rewritten using appropriate step and bounds. Namely, it may be possible to rewrite loops with a unit step control (even if extra-multiplications are thus needed in loop body). The loop below involves non-unit step control:

```
...
for(ind1=0; ind1<16; ind1+=4) {
    for(ind2=ind1; ind2<16; ind2+=4) {
        tab[ind2-ind1] = 0;
    }
}
...
```

It can be rewritten as follows:

```
...
for(ind1=0; ind1<4; ind1+=1) {
    for(ind2=ind1; ind2<4; ind2+=1) {
        tab[4*(ind2-ind1)] = 0;
    }
}
...
```

Even if this result may be contra-intuitive, the second form leads to a more efficient assembly code.

Reused variables

Description of the problem

Cross-loop induction process aims to move code out of loop nests. One of the key step consists in replacing initialization of induction variables by equivalent reset instruction located after related loop. This is based on the observation that the two pieces of code in the table below are functionally equivalent. However the one in the right-hand side is more efficient:

```
...
...
for(...) {
    i = 0 ;
    for(a=0 ; a<LC ; a++) {
        ...
        use(i) ;
        ..
        i++ ;
    }
    ...
    ...
}
...
...
i = 0;
```

Loop Restrictions

Reused variables

```
for(...) {  
    ...  
    for(a=0 ; a<LC ; a++) {  
        ...  
        use(i) ;  
        ..  
        i++ ;  
    }  
    i = i - LC ;  
    ...  
}
```

Figure 9: example of three-dimensions mixed case

As inner loop may contain a bypass test, the reset instruction must be added in the related conditional epilog. Thus, this transformation is not legal if the variable "i" is reused after inner loop, like in the figure below:

```
...  
for(...) {  
    i = 0 ;  
    for(a=0 ; a<LC ; a++) {  
        ...  
        use(i) ;  
        ..  
        i++ ;  
    }  
    ...  
    use2(i) ;  
}
```

Possible solution

One possible solution if the variable is reused after inner loop consists in creating a second variable, which is made independent from "i". This modification holds even if inner-loop count is neither

a constant nor an outer-loop invariant (case of triangular iteration space):

```
...
for(...) {
    i = 0 ;
    for(a=0 ; a<LC ; a++) {
        ...
        use(i) ;
        ..
        i++ ;
    }
    ...
    i2 = LC ;
    use2(i2) ;
}
...
```

Implicit cross-loop combination

Description of the problem

Let us consider the code below, which performs a triangular access to a linearized array:

```
...
for(i=0 ; i<MAX ; i++) {
    ...
    index = (i*GAIN) + 1 ;
    for(j=i+1 ; j<MAX ; j++) {
        ...
        tab[index]=0;
        ...
        index++ ;
        j++ ;
    }
    ...
    i++ ;
}
...
```

We notice that:

Loop Restrictions

Implicit cross-loop combination

- the index variable is inducted in inner loop,
- it also defined as a non-basic IV of outer loop,
- moreover, inner loop is bypassed once (during last iteration of outer loop, when $i=MAX-1$), but this is not a key point here.

In fact this pattern is a special case of IV redefinition. Because of the multiple and cross-loop induction, the index variable is not recognized as a cross-loop composed IV. The resulting assembly code is as follows:

```
...
DW_5
    move.w    #<1,d0                ; [17]
    move.w    #<10,d1               ; [0]
    move.w    #<1,d2                ; [16]
    doen2     #<5                   ; [0]
    dosetup2  L10                   ; [0]
    loopstart2
L10
    cmpgt.w   #<9,d0                ; [17]
    jt        L4                    ; [17]
    asr       d1,d3                 ; [0]
    move.l    d2,r0                 ; [18]
    nop                          ; [0] AGU stall
    asla      r0                    ; [18]
    adda      #>-40,sp,r1            ; [18]
    adda      r1,r0                 ; [18]
    doensh3   d3                    ; [0]
    sub       d0,d0,d3              ; [0] ;CLR
instruction
    nop                          ; [0] L_D_3
    loopstart3
L9
    move.w    d3,(r0)               ; [18]
    adda      #<2,r0                ; [19]
    loopend3
L6
L4
    add       #<4,d2                ; [15]
    sub       #<2,d1                ; [15]
    add       #<2,d0                ; [15]
    loopend2
L8
```

...

Possible solution

The solution consists in clearly separating the different components of the index cross-loop IV. Moreover the corresponding definition should be located in inner loop to allow the combination even if inner loop may be bypassed:

```
...
for(i=0 ; i<MAX ; i++) {
    ...
    index = 0 ;
    for(j=i+1 ; j<MAX ; j++) {
        invar = (i*GAIN) + 1 ;
        ...
        tab[index + invar]=0;
        ...
        index++ ;
        j++ ;
    }
    ...
    i++ ;
}
...
```

The resulting code is as follows:

```
...
.code
/* START */
.    1.move_w($d0 = 1);(50; 0)
.    1.moveu_l($r0 = &_tab2+(4));(0; 0)
.    1.move_w($r1 = 4294967268);(0; 0)
.    1.move_w($lc2 = 10);(0; 0)
.    1.dosetup(2, L18, L16);(0; 0)
.    1.move_w($d1 = 10);(0; 0)
/* START */ /* STOP */ L18:
.    10.do($d2 = $d1 - $d0);(0; 0)
.    10.do(.tst_gt($d2, 0));(50; 0)
.    10.goto (L8) .if (!$t) ;(50; 0)
.    10.move_l($lc3 = $d2);(0; 0)
.    10.dosetup(3, L17, L14);(0; 0)
.    10.do(.clr($d4));(0; 0)
```

```
/* START */ L17:
. 250.move_l($ram[$r0] = $d4) .W 226;(54; 0)
. 250.do($r0 = $r0 + 4);(55; 0)
/* START */ L14:
; (0; 0)
/* START */ L8:
. 10.do($r0 = $r0 + $r1);(0; 0)
. 10.do($d0 = $d0++);(47; 0)
. 10.do($r1 = $r1 + 4);(0; 0)
/* START */ L16:
; (0; 0)
...
```

One notices that the pointer used for memory access is stored in \$r0. It is set outside loop nest and monitored across the nest. This is possible despite the inner loop bypass test.

Conditional inner loop

Description of the problem

Cross-loop mechanism is also blocked as soon as inner loop is included in a conditional branch of outer loop body (this does not comprise the case of inner loop with bypass test, which is analyzed by the optimizer):

```
...
for(...) {
    ...
    if(condition) {
        ...
        for(j=0 ; j<LC ; j++) {
            ...
            ... tab[i][j] ... ;
            ..
            j++ ;
        }
        ...
    }
    ...
    i++ ;
}
```


...

In this case it is not possible to recombine inner and outer loop IV. The initialization of the pointer related to memory access will be located in conditional branch, instead of being moved out of the nest.

Possible solution

Unswitching outer loop, i.e. moving condition out of the nest, may overcome this limitation. Unfortunately this transformation is not always legal, especially when the loop nest is not a perfect, or as soon as the condition is not an invariant in outer loop:

```
...
if (condition) {
    for (...) {
        ...
        for (j=0 ; j<LC ; j++) {
            ...
            ... tab[i][j] ... ;
            ..
            j++ ;
        }
        ...
        i++ ;
    }
} else {
    for (...) {
        ...
        i++ ;
    }
}
...
```

Bypassed inner loop

Description of the problem

In some cases inner loop may be bypassed even if it is not located in a conditional branch. The Programmer's style may have an

incidence on such bypass tests, especially if some of the loop bounds have dynamic value (i.e. unknown at compile time). Let us consider the two examples below. We also represent the corresponding iteration spaces:

Bypass test is needed

```
...
for(i=0; i<MAX; i++) {
    for(j=0; j<=i-bound; j++) {
        ...tab[i][j]...;
        j++;
    }
    i++;
}
...
```

Bypass test can be removed

```
...
for(i=bound; i<MAX; i++) {
    for(j=0; j<=i-bound; j++) {
        ...tab[i][j]...;
        j++;
    }
    i++;
}
...
```

Figure 10: incidence of programmer style on bypass tests

From a functional point of view, the two pieces of code are equivalent, provided that loops are perfectly nested ones (i.e. with no instruction in outer loop except inner loop and induction instruction). Because of the dynamic definition of the bound value, bypass tests are always present in the initial intermediate code.

However in the left-hand-side case, the domain of the outer loop IV is wider. Inner loop is sometimes bypassed. The bypass test cannot be removed. In the right-hand-side form, inner loop is never bypassed. This is due to the shrank outer loop domain. The corresponding bypass test is then useless and can be removed. As a consequence, the right-hand-side form results in a more efficient assembly code, as shown below:

```
...
/* START */
.    1.do(.clr($d0));
.    1.move_w($d1 = $ram[&..._bound])
.    1.do($d1 = -$d1);
.    1.move_w($l2 = 10);
.    1.do_setup(2, L10, L8);
.    1.move_w($d3 = 1);
.    1.move_w($d4 = 40);
/* START */ /* STOP */ L10:
.    10.do(.tst_ge($d1, 0));
.    10.goto (L4) .if (!$t) ;
.    10.do($d2 = .tfr($d3));
.    10.do($d2 = .iadd($d2, $d1));
.    10.move_l($r0 = $d0);
.    10.moveu_l($r1 = &_tab3);
.    10.do($r1 = $r1 + $r0);
.    10.move_l($l3 = $d2);
.    10.do_setup(3, L9, L6);
.    10.do(.clr($d2));
/* START */ L9:
.    250.move_l($ram[$r1] = $d2)
.    250.do($r1 = $r1 + 4);
/* START */ L6:
/* START */ L4:
.    10.do($d1 = .iadd($d1, $d3));
.    10.do($d0 = .iadd($d0, $d4));
/* START */ L8:
...

...
/* START */ /* STOP */
.    1.move_w($d0 = $ram[&..._bound])
.    1.do(.cmp_gt($d0, 10-1));
.    1.goto (L2) .if ($t) ;
.    1.move_w($d1 = 10);
.    1.do($d2 = $d1 - $d0);
.    1.move_w($d3 = 65535);
.    1.do($d4 = .tfr($d0));
.    1.do($d4 = .imac($d3, $d0));
.    1.do($d4 = .sxt_w($d4));
.    1.do($d4 = $d4++);
```

Loop Restrictions

Bypassed inner loop

```
.      1.do($d0 = .impy($d0, 40));
.      1.move_l($r0 = $d0);
.      1.moveu_l($r1 = &_tab3);
.      1.do($r0 = $r0 + $r1);
.      1.do(.clr($d5));
.      1.do($d4 = .asl_imm($d4, 2));
.      1.do($d4 = .sxt_w($d4));
.      1.move_w($d6 = 40);
.      1.do($d7 = $d6 - $d4);
.      1.move_l($lc2 = $d2);
.      1.dosetup(2, L10, L8);
.      1.move_w($d0 = 1);
.      1.do(.clr($d9));
/* START */ L10:
.      25.do($d8 = .tfr($d0));
.      25.do($d8 = .iadd($d8, $d5));
.      25.move_l($lc3 = $d8);
.      25.dosetup(3, L9, L6);
/* START */ L9:
.      625.move_l($ram[$r0] = $d9)
.      625.do($r0 = $r0 + 4);
/* START */ L6: (0; 0)
.      25.move_l($r2 = $d7);
.      25.do($r0 = $r0 + $r2);
.      25.do($d5 = .iadd($d5, $d0));
.      25.do($d7 = $d7 - 4);
/* START */ L8:
...
```

Figure 11: resulting assembly code

The optimized code on the right-hand-side is larger but it is more efficient. If MAX equals 10, bound equals 7, and tab is a 10x10 array, then the number of cycles decreases from 187 (right-hand-side) to 134 for such a simple example.

Possible solution

The first form should be avoided if possible.

Limitations of Sequential Accesses and Packing

- [Aliasing and conflicting interleaved read/write accesses](#)
- [Aliasing and interprocedural effect](#)
- [Missing initial alignment](#)

Aliasing and conflicting interleaved read/write accesses

Description of the problem

Let us now consider a function that receives two pointers as arguments. Let us assume that those pointers are used to copy the content of one array in the other one, like in the code below:

```
...
void copy_fct(short* A1, short* A2)
{
    #pragma noline
    #pragma align *A1 8;
    #pragma align *A2 8;

    for(i=0, j=0; i<MAX; i+=2, j+=2) {
        A2[i]    = A1[j];
        A2[i+1]  = A1[j+1];
    }
}
...
```

The equivalent sequence of basic instructions in loop would be as follows:

```
...
for(i=0, j=0; i<MAX; i+=2, j+=2) {
    tmp1    = A1[j];/* instruction 1 */
    A2[i]    = tmp1;/* instruction 2 */
    tmp2    = A1[j+1];/* instruction 3 */
    A2[j+1]  = tmp2;/* instruction 4 */
}
...
```

Loop Restrictions

Aliasing and conflicting interleaved read/write accesses

Access packing aims to bring the code to the following form, where read accesses to A1 are packed together on one side, and write accesses to A2 are packed on the other side:

```
...
    for(i=0, j=0; i<MAX; i+=2, j+=2) {
        tmp1    = A1[j]; /* instruction 1 */
        tmp2    = A1[j+1]; /* instruction 3 */
        A2[i]    = tmp1; /* instruction 2 */
        A2[j+1] = tmp2; /* instruction 4 */
    }
...
```

Unfortunately, in this case, pointers A1 and A2 are provided as input arguments. Nothing precise is known concerning A1 and A2, which thus belong to the same alias class. In other words A1[i] and A2[j+1] may point the same memory location. Thus, moving instruction 3 before instruction 2 is not legal, because instruction 3 may redefine the content of A2[j+1]. Packing accesses in this case is not possible.

Possible solution

The code may be transformed so as to make this packing legal. In this case the transformation simply consists in avoiding interleaved accesses. The programmer then assumes that it is legal to do, due to the way the function is used. The loop is rewritten as follows:

```
...
    for(i=0, j=0; i<MAX; i+=2, j+=2) {
        tmp1    = A1[j]; /* instruction 1 */
        tmp2    = A1[j+1]; /* instruction 3 */
        A2[i]    = tmp1; /* instruction 2 */
        A2[j+1] = tmp2; /* instruction 4 */
    }
...
```

Aliasing and interprocedural effect

Description of the problem

Let us restart from the code above. The only modification is a new function call, which is located between the two read accesses. This function receives pointer A1 as input argument:

```
...
int fct(short* tab)
{
    #pragma noline
    return(tab[0]);
}

void copy_fct(short* A1, short* A2)
{
    #pragma noline
    #pragma align *A1 8;
    #pragma align *A2 8;

    short tmp1, tmp2;

    for(i=0, j=0; i<MAX; i+=2, j+=2) {
        tmp1    = A1[j];
        c = fct(A1);
        tmp2    = A1[j+1];
        A2[i]    = tmp1;
        A2[j+1] = tmp2;
    }
    ...
}
```

As function fct receives the pointer as input argument, it is likely to modify its value or the content of related memory locations.

Possible solution

Moving function call out of the sequence is not the appropriate solution: the pointer may still be redefined and the condition on preservation of alignment may thus be violated. The only solution consists in:

- either declaring that function fct has no side effect thanks to the related nosideeffects pragma. This prevents the compiler from choosing the default worst case assumption concerning aliasing,
- or inlining function fct, so as to make data dependencies explicit.

In many cases, function calls may induce this kind of ambiguity.

Missing initial alignment

Description of the problem

We still consider a more complex function, where the sequence of accesses is performed in a triangular loop:

```
...
void fct(short* A1)
{
    #pragma noline
    #pragma align *A1 8;

    short tmp1, tmp2;

    for(i=0; i<MAX; i+=2) {
        for(j=i; j<MAX; j++) {
            tmp1    = A1[j];
            tmp2    = A1[j+1];
            ...
            /*tmp1 and tmp2 assumed to be used in
loop*/
            ...
        }
    }
    ...
}
```

In this case the compiler cannot detect that the condition on alignment is satisfied. Namely, j is used as an index. It is defined using a dynamic initial value (derived from outer loop IV). Thus access packing cannot be performed.

Possible solution

One possible solution consists in adding one function where inner loop is implemented, and where initial array alignment is specified thanks to a pragma:

```
...
void fct_inner(short* A1)
{
    #pragma noline
    #pragma align *A1 8;

    short tmp1, tmp2;

    for(j=i; j<MAX; j++) {
        tmp1    = A1[j];
        tmp2    = A1[j+1];
        ...
        /*tmp1 and tmp2 assumed to be used in loop*/
        ...
    }
    ...
}
...
void fct(short* A1, short* A2)
{
    #pragma noline

    for(i=0; i<MAX; i+=2) {
        fct_inner(A1);
    }
    ...
}
...
```

Case Study: G729 cor_h Function

- [Purpose and content of this chapter](#)
- [Structure of function loop nests](#)
- [Restrictions and solutions](#)

- [Result](#)

Purpose and content of this chapter

We now make use of a function found in a real-life signal processing application, to illustrate the restrictions of the high-level optimizer. We also present a possible way to rewrite this function so as to fit the compiler constraints.

For the sake of readability, the example corresponds with a subpart of the real G729 vocal coder `cor_h` function (one of the loop nest).

Structure of function loop nests

The loop-nest used as example is as follows:

```
...
i1= 0;
i2= 1;
p3 = rri2i3- 2;
p2 = rri1i2- 1;
p1 = rri0i1- 1;
p0 = rri0i4- 4;
l_fin = MSIZE;

for(k=0; k<NB_POS; k++){
    p3+= 2*l_fin;
    p2+= l_fin;
    p1+= l_fin;
    p0+= 2*l_fin;

    cor0= 0;
    for(i= k+1; i< NB_POS; i++) {
        cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
        cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
        *p3=mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

        cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
        *p2 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

        cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
```

```
    *p1 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

    cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
    *p0 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

    p3 -= 2*(NB_POS+ 1);
    p2 -= NB_POS+ 1;
    p1 -= NB_POS+ 1;
    p0 -= 2*(NB_POS+ 1);
}
cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
*p3 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
*p2 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
*p1 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

l_fin-= NB_POS+ 1;
p0+= 2*(NB_POS- 1);
i1-= (NB_POS-k-1)*5+4;
i2-= (NB_POS-k-1)*5+4- STEP;
}
...
```

Restrictions and solutions

First step: removing reused IV

a. Problem(s)

The first remark we can make concerns inner loop IV *i1* and *i2*. Those variables are reused in outer loop after inner loop. As a consequence the cross-loop mechanism cannot apply the "reset"

transformation. Variables i1 and i2 are also both inducted and redefined (reset) in outer loop. The redefinition involves of another local IV: k. All those features constitute restrictions for the high-level optimizer.

b. Solution(s)

The solution consists in using a different set of variable for memory access in inner and outer loop. We thus create two new induction variables to be used in outer loop: i1b and i2b. Those variables are set to the appropriate value. Subsequently, variables i1 and i2 are no longer reset in outer loop. They are now defined instead. The corresponding code is as follows (modified parts in bold font):

```
...
i1= 0;
i2= 1;
p3 = rri2i3- 2;
p2 = rri1i2- 1;
p1 = rri0i1- 1;
p0 = rri0i4- 4;
l_fin = MSIZE;
for(k=0; k<NB_POS; k++){
    p3+= 2*l_fin;
    p2+= l_fin;
    p1+= l_fin;
    p0+= 2*l_fin;

    cor0= 0;
    i1 = 0; i2 = (k*STEP)+1; /*i1 and i2 now set before inner loop*/
    for(i= k+1; i< NB_POS; i++) {
        cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
        cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
        *p3=mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

        cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
        *p2 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

        cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
        *p1 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));
```

```
cor0 = L_mac(cor0, h[i1], h[i2]); i1++; i2++;
*p0 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2]));

p3 -= 2*(NB_POS+ 1);
p2 -= NB_POS+ 1;
p1 -= NB_POS+ 1;
p0 -= 2*(NB_POS+ 1);
}
ilb=5*(NB_POS-(k+1));/*new variables ilb and i2b set here*/
i2b=5*(NB_POS-(k+1))+((k*STEP)+1);

cor0 = L_mac(cor0, h[i1b], h[i2b]); i1b++; i2b++; /*new
variables ilb and i2b used*/
cor0 = L_mac(cor0, h[i1b], h[i2b]); i1b++; i2b++;
*p3 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1b],
Sign_Dn[L_SUBFR-i2b]));

cor0 = L_mac(cor0, h[i1b], h[i2b]); i1b++; i2b++;
*p2 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1b],
Sign_Dn[L_SUBFR-i2b]));

cor0 = L_mac(cor0, h[i1b], h[i2b]); i1b++; i2b++;
*p1 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1b],
Sign_Dn[L_SUBFR-i2b]));

l_fin-= NB_POS+ 1;
p0+= 2*(NB_POS- 1);}
...
```

Second step: removing implicit cross-loop combination

a. Problem(s)

This new form still contains one conflicting pattern: inner loop IV i2 is set using outer loop IV k. This kind of implicit cross-loop combination is not handled by the compiler.

b. Solution(s)

The solution consists in clearly separating each subpart of the combination: inner-loop invariant part (i.e. outer loop IV) on one side, inner loop IV on the other side. A new invar variable is created to store invariant part of indices. Initial value of i2 and actual

indices are modified appropriately. The code is thus transformed as follows:

```
...
i1= 0;
i2= 1;
p3 = rri2i3- 2;
p2 = rri1i2- 1;
p1 = rri0i1- 1;
p0 = rri0i4- 4;
l_fin = MSIZE;

for(k=0; k<NB_POS; k++){
    p3+= 2*l_fin;
    p2+= l_fin;
    p1+= l_fin;
    p0+= 2*l_fin;

    cor0= 0;
    i1 = 0; i2 = 0;
    for(i= k+1; i< NB_POS; i++) {
        invar = (k*STEP)+1;

        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        *p3 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2-invar]));

        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        *p2 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2-invar]));

        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        *p1 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2-invar]));

        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        *p0 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2-invar]));

        p3 -= 2*(NB_POS+ 1);
        p2 -= NB_POS+ 1;
        p1 -= NB_POS+ 1;
```

```
    p0 -= 2*(NB_POS+ 1);
}
i1b=5*(NB_POS-(k+1));
i2b=5*(NB_POS-(k+1))+((k*STEP)+1);

cor0 = L_mac(cor0, h[i1b], h[i2b]); i1b++; i2b++;
cor0 = L_mac(cor0, h[i1b], h[i2b]); i1b++; i2b++;
*p3 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1b],
Sign_Dn[L_SUBFR-i2b]));

cor0 = L_mac(cor0, h[i1b], h[i2b]); i1b++; i2b++;
*p2 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1b],
Sign_Dn[L_SUBFR-i2b]));

cor0 = L_mac(cor0, h[i1b], h[i2b]); i1b++; i2b++;
*p1 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1b],
Sign_Dn[L_SUBFR-i2b]));

l_fin-= NB_POS+ 1;
p0+= 2*(NB_POS- 1);}
...
```

Third and final step: removing redefined IV in outer loop

a. Problem(s)

The last restrictive pattern concerns i1b and i2b variables. Those variables are redefined IV. Namely:

- they are initialized like non-basic IV derived from k (i1b=5*(NB_POS-(k+1)),
- they are also inducted like multi-step IV (i1b++ occurs three times).

The optimizer cannot handle such patterns.

b. Solution(s)

The solution consists in:

- removing multi-step induction on both i1b and i2b,
- modifying indices in memory accesses (i1b successively replaced by i1b, i1b+1, i1b+2, ...).

The code is thus transformed as follows:

Loop Restrictions

Restrictions and solutions

```
...
i1= 0;
i2= 1;
p3 = rri2i3- 2;
p2 = rri1i2- 1;
p1 = rri0i1- 1;
p0 = rri0i4- 4;
l_fin = MSIZE;

for(k=0; k<NB_POS; k++){
    p3+= 2*l_fin;
    p2+= l_fin;
    p1+= l_fin;
    p0+= 2*l_fin;

    cor0= 0;
    i1 = 0; i2=0;
    for(i= k+1; i< NB_POS; i++) {
        invar = (k*STEP)+1;

        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        *p3 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2-invar]));

        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        *p2 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2-invar]));

        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        *p1 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2-invar]));

        cor0 = L_mac(cor0, h[i1], h[i2+invar]); i1++; i2++;
        *p0 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1],
Sign_Dn[L_SUBFR-i2-invar]));

        p3 -= 2*(NB_POS+ 1);
        p2 -= NB_POS+ 1;
        p1 -= NB_POS+ 1;
        p0 -= 2*(NB_POS+ 1);
    }
}
```



```
    i1b=5*(NB_POS-(k+1));
    i2b=5*(NB_POS-(k+1))+((k*STEP)+1);

    cor0 = L_mac(cor0, h[i1b], h[i2b]);
    cor0 = L_mac(cor0, h[i1b+1], h[i2b+1]);
    *p3 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1b-2],
Sign_Dn[L_SUBFR-i2b-2]));

    cor0 = L_mac(cor0, h[i1b+2], h[i2b+2]);
    *p2 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1b-3],
Sign_Dn[L_SUBFR-i2b-3]));

    cor0 = L_mac(cor0, h[i1b+3], h[i2b+3]);
    *p1 = mult(extract_h(cor0), mult(Sign_Dn[L_SUBFR-i1b-4],
Sign_Dn[L_SUBFR-i2b-4]));

    l_fin-= NB_POS+ 1;
    p0+= 2*(NB_POS- 1);
}
...
```

Result

The actual `cor_h` function contains four loop nests with such a similar structure. When those loop nests are transformed as proposed above, the overall gain is about 2000 cycles.

Index

A

- abort environment function 211
- abs integer arithmetic function 210
- abs_s intrinsic function 69, 216
- acos trigonometric function 203
- add intrinsic function 69, 216
- align #pragma 76
- align pragma 77
- Alignment
 - bit-fields 62
 - variables 86
- ansi shell option 30, 39
- Application configuration file 173
 - binding section 176
 - overlay section 177
 - schedule section 174
- Application entry point 162
- arch shell option 31
- Arithmetic
 - fixed point 61
 - floating point 61
 - fractional 64
 - integer 64
- asctime time function 214
- asin trigonometric function 203
- asm statement 94
- Assembly functions 100
- Assembly instruction inlining
 - asm statement 94
- Assembly instructions
 - inlining sequence 94
 - inlining single instruction 93
- atan trigonometric function 203
- atan2 trigonometric function 203
- atexit environment function 211
- atof string conversion function 210
- atoi string conversion function 210
- atol string conversion function 210

B

- Bare board startup 160
- Bare board startup code 160
- Basic block 108, 136

- be shell option 32, 45, 59, 61, 63
- Big memory model 163
- Big-endian mode 45
- Binding section
 - application configuration file 176
- Bit-fields 62
- BitReverseUpdate intrinsic function 73, 216
- bss_seg_name pragma 77
- Built-in intrinsic functions 216

C

- C environment startup 161
- C environment startup code 160
- C language
 - dialects 45
 - extensions 46
 - K&R 51
 - PCC 51
- C language options 39
- C shell option 29, 36
- c shell option 29, 34
- Call tree 174
- call_conv pragma 77
- Calling convention
 - stack-based 180
 - stack-less 182
- calloc memory allocation function 209
- ceil function 204
- cfe shell option 29, 33
- Character typing 196
- clearerr stream function 206
- clock time function 214
- clock_t time function 214
- Code
 - linear 108
 - migrating from other environments 227
 - parallelized 109
 - transformations 107
- Command file 28, 34
- Command line 27
- Common subexpression elimination 126
- Comparison functions 213
- Compatibility clause 178

Index

- Compilation process 13, 20
- Composed variable loop 120
- Concatenation functions 212
- Conditional execution 135
- Configuration
 - memory map 168
 - startup code 162
- Constant folding 127
- Control options 28
- Conversion functions 197
- Copying functions 212
- cos trigonometric function 203
- cosh hyperbolic function 203
- Cross-file optimization 110, 113, 140
- crt shell option 32, 44
- ctime time function 214
- ctype.h library 195, 196

D

- D shell option 29, 36
- D_add intrinsic function 70, 216
- D_cmpeq intrinsic function 70
- D_cmpgt intrinsic function 70
- D_extract_h intrinsic function 71, 216
- D_extract_l intrinsic function 71, 216
- D_mac intrinsic function 70, 216
- D_msu intrinsic function 70, 216
- D_mult intrinsic function 70, 217
- D_round intrinsic function 71, 217
- D_sat intrinsic function 71, 217
- D_set intrinsic function 71, 217
- D_sub intrinsic function 70, 217
- Data allocation
 - static 168
- Data types 57
 - bit-fields 62
 - character 58
 - double precision fractional 67
 - extended precision fractional 67
 - floating point 61
 - fractional long 67
 - fractional representation 61
 - fractional short 67
 - integer 59
 - pointers 62
- data_seg_name pragma 77

- dc shell option 31, 42
- de shell option 31, 41
- Dead assignment
 - elimination 128
- Dead code
 - elimination 128
- Dead storage
 - elimination 128
- debug intrinsic function 72, 217
- debugev intrinsic function 72, 217
- default_call_conv pragma 77
- Delay slots 132
- Dependencies
 - between instructions 109
- Dependency 115
- di intrinsic function 73, 217
- Dialects
 - C language 45
- difftime time function 214
- div integer arithmetic function 210
- div_s intrinsic function 69, 217
- dL shell option 31, 41
- dL1 shell option 31, 41
- dL2 shell option 31, 42
- dL3 shell option 31, 42
- dm shell option 31, 41
- do shell option 31, 41
- Double precision 68
- DSP56600 compiler
 - differences 228
 - header file 227, 228
 - migrating code 227, 228
- dx shell option 31, 42
- Dynamic loop 118
- Dynamic memory allocation 167
- dynamic pragma 76

E

- E shell option 29, 33
- ei intrinsic function 72, 217
- Elimination
 - dead assignment 128
 - dead code 128
 - dead storage 128
 - jump-to-jump 127
 - subexpression 126

EndBitReverse intrinsic function 73, 217
Entry points 174
Environment functions 211
Environment variables 27
Execution sets 108
 parallelized 114
Execution units 108
exit environment function 211
exp function 203
Exponential functions 203
Extended 71
Extended precision 67
Extensions 25, 37
 C language 46
external #pragma 76
External function 80
external pragma 78
extract_h intrinsic function 70, 217
extract_l intrinsic function 70, 217

F

-F shell option 29, 34
fabs function 204
fclose stream function 206
feof stream function 206
ferror stream function 207
fflush I/O function 209
fgetc input function 206
fgetpos stream function 207
fgets output function 208
File extensions 25, 37
File types 25
Finalization code 159, 162
float.h library 195, 197
Floating point arithmetic 61
Floating point characteristics 197
Floating point math 202
floor function 204
fmod function 204
fopen I/O function 209
fprintf output function 208
fputc output function 208
fputs output function 208
Fractional
 arithmetic 64
 constants 68

 representation 61
 values 68
fread input function 206
free memory allocation function 209
freopen stream function 207
frexp function 203
fscanf input function 206
fseek stream function 207
fsetpos stream function 207
ftell stream function 207
Function inlining 125
Functions
 built-in intrinsic 216
 comparison 213
 concatenation 212
 conversion 197
 copying 212
 environment 211
 exponential 203
 external 80
 hyperbolic 203
 I/O 208
 input 206
 integer arithmetic 209
 intrinsic 67, 69, 216
 logarithmic 203
 memory allocation 209
 multibyte character 211
 output 207
 power 204
 pseudo random number generation 211
 search 213
 searching 210
 sorting 210
 stream 206
 string 212
 string conversion 210
 testing 196
 time 214
 trigonometric 202
fwrite output function 208

G

-g shell option 30, 39
General utilities 209
getc input function 206
getchar input function 206

Index

getenv environment function 211
gets output function 208
Global variables 178
gmtime time function 215
Guidelines
 optimizer 141

H

-h shell option 29, 35
Hardware loops 194
Hardware registers
 initialization 160
Header file
 TI6xx compiler 227
Heap 168
Hyperbolic functions 203

I

-I shell option 29, 37
I/O functions 208
I/O services
 low level 162
 termination 162
illegal intrinsic function 73, 217
Include files 37
init_seg_name pragma 77
InitBitReverse intrinsic function 73, 217
Initialization
 M registers 160
 status registers 161
 variables 44, 162
Initialization code 159, 161
Initializing variables with fractional values 68
inline #pragma 76
inline pragma 76
Inlining 78, 125
 sequence of assembly instructions 94
 single assembly instruction 93
Input file extension 37
Input functions 206
Instruction scheduling 131
Instruction transformations 108
Integer arithmetic 65
Integer arithmetic functions 209
Integer characteristics 201
interrupt #pragma 76

Interrupt entry 176, 193
Interrupt handler 81, 174, 193
interrupt pragma 78
Interrupt vector 160, 176, 193
Interrupts 161
Intrinsic functions 67
 architecture primitives 72
 assembly instruction architecture
 primitives 72
 bit reverse addressing 73
 double precision fractional arithmetic 70
 fractional arithmetic 69
 fractional arithmetic with guard bits 71
 long fractional arithmetic 70

Invariant code loop 126
isalnum testing function 196
isalpha testing function 196
isctrl testing function 196
isdigit testing function 196
isgraph testing function 196
islower testing function 196
ISO libraries 195
isprint testing function 196
ispunct testing function 196
isspace testing function 197
isupper testing function 197
isxdigit testing function 197

J

Jump-to-jump elimination 127

K

K&R mode 51
-kr shell option 30, 39

L

L_abs intrinsic function 70, 217
L_add intrinsic function 218
L_deposit_h intrinsic function 70, 218
L_deposit_l intrinsic function 70, 218
L_mac intrinsic function 69, 218
L_max intrinsic function 70, 218
L_min intrinsic function 70, 218
L_msu intrinsic function 69, 218
L_mult intrinsic function 218

L_negate intrinsic function 70, 218
L_rol intrinsic function 72, 219
L_ror intrinsic function 72, 219
L_sat intrinsic function 70, 219
L_shl intrinsic function 70, 219
L_shr intrinsic function 70, 219
L_shr_r intrinsic function 70, 219
L_sub intrinsic function 70, 219
labs integer arithmetic function 210
L-add intrinsic function 70
ldexp function 203
ldiv integer arithmetic function 210
Libraries
 ISO 195
 non-ISO 196
limits.h library 195, 201
Linear code 108
Linker command file 161, 164
Listing files 41
Little-endian 45
Little-endian mode 63
Little-endian representation 59, 61
L-mult intrinsic function 70
locale.h library 195, 202
localeconv locales function 202
Locales functions 202
localtime time function 215
log function 203
log10 function 203
Logarithmic functions 203
Logical memory 170
Loop
 composed variable 120
 dynamic 118
 multi-step 119
 simple 117
 square 121
 transformations 116
Loop count 84
loop_count #pragma 76
loop_count pragma 78
Loops
 hardware 194
Low level transformations (LLT) 129

M

M registers
 initialization 160
 value 194
-M shell option 29, 36
-ma shell option 32, 44
mac_r intrinsic function 69, 219
Machine configuration file 170
Macros 36
 fractional values 68
 predefined 91
 preprocessor 36
Main entry point 174
malloc memory allocation function 209
mark intrinsic function 72, 219
math.h library 195, 202
max intrinsic function 69, 219
-mb shell option 32, 44
-mc shell option 32, 44
-mem shell option 32, 44
memchr search function 213
memcmp comparison function 213
memcpy copying function 212
memmove copying function 212
Memory
 logical 170
 mode 44, 164
 physical 170
Memory allocation
 dynamic 167
 functions 209
Memory layout
 default 165
Memory map
 configuration 168
 default values 166
 initialization 161
Memory model
 big 163
 small 164
Memory space 170
memset function 214
Messages 41
-MH shell option 29, 36
Migrating code 227
min intrinsic function 70, 219

Index

mktime time function 214

Mode

 K&R/PCC 51

modf function 203

modulo addressing example 235

mpysu intrinsic function 72, 220

mpyus intrinsic function 72, 220

mpyuu intrinsic function 72, 220

-mrom shell option 32

msu_r intrinsic function 69, 220

mult intrinsic function 69, 220

mult_r intrinsic function 69, 220

Multibyte character functions 211

Multiple execution units 108

Multi-step loop 119

N

-n shell option 31, 42

negate intrinsic function 69, 220

noinline #pragma 76

noinline pragma 76

Non-cross file optimization 23

Non-ISO libraries 196

Nonlocal jumps 204

norm_l intrinsic function 70, 221

norm_s intrinsic function 69, 221

O

-o shell option 30, 38

-O0 shell option 30, 111

-O1 shell option 30, 111, 115

-O2 shell option 30, 111, 129

-O3 shell option 30, 111

-Og shell option 31, 111, 140

opt_level pragma 76

Optimization

 cross file 12, 22, 110, 113, 140

 for size 113, 139

 levels 110

 non-cross file 23, 24

 options 110

 target independent 114, 115

 target specific 114, 129

Optimizer

 guidelines 141

 invoking 112

Options

 C language 39

 control 28

 extensions 37

 messages 41

 output files 38

 shell 29

-Os shell option 30, 111, 139

Output files 38

Output functions 207

Overlay section

 application configuration file 177

Overlay specification 175

P

Parallelized code 109

Parallelized execution sets 114

Passing options 40

perror output function 208

pgm_seg_name pragma 77

Physical memory 170

Pipeline restrictions 132

Pointers 62

Post-increment detection 137

pow function 204

Power functions 204

Pragmas

 #pragma align 76, 87

 #pragma external 76, 81

 #pragma inline 76

 #pragma interrupt 81

 #pragma loop count 76, 85

 #pragma noline 76, 79

 #pragma profile 76, 83, 84

 #pragma save 76

 #pragma save_ctxt 79

 placement 75

 syntax 75

Predefined macros 91

Prefix grouping 139

Preprocessing options 35

Preprocessor macros 36

printf output function 208

Process time 215

profile #pragma 76

Profile value 83

prototype.h 196

-
- prototype.h library 196, 216
 - Pseudo random number generation functions 211
 - putc output function 208
 - putchar output function 208
 - puts output function 208
 - Q**
 - q shell option 31, 42
 - R**
 - r shell option 30, 38
 - rand pseudo random number generation function 211
 - realloc memory allocation function 209
 - remove stream function 207
 - rename stream function 207
 - Reporting 42
 - Reset interrupt vector 160
 - rewind stream function 207
 - rom_seg_name pragma 77
 - round intrinsic function 69, 221
 - Runtime
 - environment 159
 - startup code 159
 - S**
 - S shell option 29, 34
 - safe_mod pragma 76
 - saturate intrinsic function 70, 221
 - save_ctxt #pragma 76
 - sc shell option 30, 40
 - scanf input function 206
 - Schedule section
 - application configuration file 174
 - Search functions 213
 - Searching functions 210
 - set2cnvrm intrinsic function 72, 221
 - set2crm intrinsic function 72, 221
 - setbuf stream function 207
 - setjmp.h library 195, 204
 - setlocale locales function 202
 - setnosat intrinsic function 72, 221
 - setsat32 intrinsic function 72, 221
 - setvbuf stream function 207
 - Shell 13, 19
 - Shell command file 34
 - Shell options
 - behavior control 29
 - C language 30
 - file extension override 30
 - hardware model and configuration 31
 - optimization pragma and code 30
 - output filename and location 30
 - output of listing files and messages control 31
 - pass-through 31
 - preprocessing 29
 - stop processing 33
 - summary 29
 - shl intrinsic function 69, 221
 - shr intrinsic function 69, 222
 - shr_r intrinsic function 69, 222
 - Signal handling 204
 - signal.h library 193, 195, 204
 - Simple loop 117
 - sin trigonometric function 203
 - sinh hyperbolic function 203
 - Small memory model 163
 - Software pipelining 133
 - Sorting functions 210
 - Space optimization 113, 139
 - Speculative execution 136
 - sprintf output function 208
 - sqrt function 204
 - Square loop 121
 - strand pseudo random number generation function 211
 - sscanf input function 206
 - Stack
 - frame 183
 - memory allocation 166
 - pointer 167, 180
 - space 184
 - start address 161, 167
 - Stack-based
 - calling convention 180
 - Stack-less
 - calling convention 182
 - Standard definitions 205
 - Startup code 159
 - bare board 160
 - C environment 160
 - configuration 162

Index

Static data allocation 168
Status registers
 default settings 161
 initialization 161
stdarg.h library 195, 205
stddef.h library 195, 205
stderr stream function 207
stdin stream function 207
stdio.h library 195, 206
stdlib.h library 195, 209
stdout stream function 207
stop 33
stop intrinsic function 72, 222
strcat concatenation function 213
strchr search function 213
strcmp comparison function 213
strcoll comparison function 213
strcpy copying function 212
strcspn search function 213
Stream functions 206
Strength reduction 116
strerror function 214
strftime time function 214
String conversion functions 210
String functions 212
string.h library 195, 212
strlen function 214
strncat concatenation function 213
strncmp comparison function 213
strncpy copying function 212
strpbrk search function 213
strrchr search function 213
strspn search function 214
strstr search function 214
strtod string conversion function 210
strtok search function 214
strtol string conversion function 210
strtoul string conversion function 210
strxfrm comparison function 213
sub intrinsic function 69, 222
Subexpression elimination 126
Symbolic labels 104
System context 79

T

tan trigonometric function 203
tanh hyperbolic function 203
target architecture 43
Target-independent optimizations 115
Target-specific optimizations 129
Target-specific optimizations 114
Target-specific peephole 138
Task entry point 174
Termination
 I/O services 162
Testing functions 196
TI6xx compiler
 header file 227
 migrating code 227
Time constant 215
time function 214
Time functions 214
time.h library 195, 214
time_t time function 215
Timer 161
tolower conversion function 197
toupper conversion function 197
Transformations
 loop 116
trap intrinsic function 72, 222
Trigonometric functions 203

U

-U shell option 29, 36
ungetc I/O function 209
-usc shell option 30, 40

V

-v shell option 31, 42
Variable arguments 205
Variables
 alignment 86
 initialization 162
vfprintf output function 208
vprintf output function 208
vsprintf output function 208

W

-w shell option 31, 42

wait intrinsic function 72, 222
-Wall shell option 31, 43
WORD16 macro 68
WORD32 macro 68
Word40 extended precision fractional 67
Word64 double precision fractional 68

X

X_abs intrinsic function 71, 222
X_add intrinsic function 71, 222
X_cmpeq intrinsic function 72
X_cmpgt intrinsic function 72
X_extend intrinsic function 72, 222
X_extract_h intrinsic function 71, 222
X_extract_l intrinsic function 71, 222
X_mac intrinsic function 71, 222
X_msu intrinsic function 71, 222

X_mult intrinsic function 71, 223
X_norm intrinsic function 71, 223
X_or intrinsic function 71, 223
X_rol intrinsic function 71, 223
X_ror intrinsic function 71, 223
X_round intrinsic function 71, 223
X_sat intrinsic function 71, 223
X_set intrinsic function 71, 223
X_shl intrinsic function 71, 223
X_shr intrinsic function 71, 223
X_sub intrinsic function 71, 223
X_trunc intrinsic function 71, 223
-Xasm shell option 31, 40
-xasm shell option 30, 37
-xc shell option 30, 37
-Xlnk shell option 31, 40
-xobj shell option 30, 37

